5-1-2017

A Fast and Efficient Method for Power Distribution Network Reconfiguration

Aaron Jordan Ekstrand
Southern Illinois University Carbondale, aaronekstrand@gmail.com

A FAST AND EFFICIENT METHOD FOR POWER DISTRIBUTION NETWORK RECONFIGURATION

by

Aaron Ekstrand

B.A., Cornell College, 2013

A Thesis

Submitted in Partial Fulfillment of the Requirements for the

Master of Science Degree

Department of Engineering

In the Graduate School

Southern Illinois University Carbondale

May 2017

THESIS APPROVAL

A FAST AND EFFICIENT METHOD FOR POWER DISTRIBUTION NETWORK RECONFIGURATION

By

Aaron Ekstrand

A Thesis Submitted in Partial

Fulfillment of the Requirements

for the Degree of

Master of Science

in the field of Electrical & Computer Engineering

Approved by:

Dr. Dimitri Kagaris, Chair

Dr. Ning Weng

Dr. Constantine Hatziadoniu

Graduate School

Southern Illinois University Carbondale

November 2016

AN ABSTRACT OF THE THESIS OF

Aaron Ekstrand, for the Master of Science degree in Electrical & Computer Engineering, presented on November 3, 2016, at Southern Illinois University Carbondale.

TITLE: A FAST AND EFFICIENT METHOD FOR POWER DISTRIBUTION NETWORK RECONFIGURATION

MAJOR PROFESSOR: Dr. Dimitri Kagaris

In any power distribution network, there is the risk of power generator failure or other types of faults with components and wires. In such circumstances, there may suddenly be more loads in the network than there is power to supply them.

This thesis investigates a fast, automated solution to this problem. First, we present a distributed algorithm for gathering the information about the topology of a network. This is based on a variation of the Bellman-Ford algorithm. Then we present two algorithms for automatically assigning power sources to loads according to the ranking of importance (priority) that those loads are given.

The first power assignment algorithm is based on an integer linear programming formulation, and the second is a heuristic. As such, they will be referred to as the ILP Solver and the Heuristic Solver, respectively. The ILP Solver always returns the optimal assignment of generators to loads, but is much slower, and in networks that are very large, it is infeasible to wait for a solution. The Heuristic Solver returns similar, sometimes identical results, depending on the complexity of the topology, and it finds its solution orders of magnitude faster than the

ILP Solver. Experimental evaluation is given to compare the performance of the proposed Heuristic Solver to the ILP Solver, which eventually returns the ideal solution.

TABLE OF CONTENTS

LIST OF TABLES

TABLE PAGE

LIST OF FIGURES

LIST OF PSEUDOCODE BLOCKS

INTRODUCTION & MOTIVATION

Power distribution networks are circuits designed to deliver electrical power from one or more electrical power generators (sources) to one or more loads that consume energy. There may be many types of components in these networks—transformers, circuit breakers, capacitors, and many others. The scope of this research only requires the understanding of a few components: sources, loads, lines, and switches.

Sources and loads are connected by lines, which conduct power from the sources that generate the power to the loads that consume it. These lines may be gated by switches, which are breaks in the line that may be freely opened or closed in order to prevent or permit current from being conducted along that channel.

A power distribution network is typically a system of power plants, substations, and power lines that supply electricity to homes and businesses in industrialized societies. Power distribution networks also appear in other contexts. For example, an aircraft typically has multiple power generators that supply all loads on board, from the navigation system to the coffee maker. A hospital may have multiple gas-powered generators, which keep the hospital running in case of a blackout. Having the monitors go out in the middle of a surgery would be fairly catastrophic.

Ideally, the sources in a network generate more than enough power to feed all the loads that are needed at any one time, but failures are always a possibility. Sources fail, lines short out. There might not be enough power for all of the loads. If one is on an airplane and the coffeemaker suddenly stops working, the consequences are at worst a minor annoyance. However, if the

navigation system stops working, then there is some real trouble. That is why in an emergency, it's an important problem to optimize use of the sources and lines that are still working to deliver power to the most important loads.

Power distribution network reconfiguration is a widely-researched topic in electrical engineering. Power configuration is the initial establishment of switch states in a network. Where distributed generators are placed in a network can be as impactful as configuration [1], but we're going to focus on optimizing configuration of established power distribution networks, not on designing them. Specifically, we are going to focus on reconfiguration, which means establishing new switch states, as is necessary in the case that some fault changes the topology of the network.

While one might think that optimization of power flow in an established network is static, "This traditional view does not describe transmission assets as assets that operators have the ability to control. However, it is acknowledged, both formally and informally, that system operators can and do change the grid topology to improve voltage profiles, increase transfer capacity, and even improve system reliability" [2]. Making use of manual controls to decide how power should be directed can have expensive and potentially dire real-world costs, compared to relying on automated solutions. Human error and human computation time are often unacceptable for sensitive applications. For example, for the purpose of reliability, efficiency, safety, as well as other benefits, automated control is an essential part of Smart Grids [3], which industrialized countries march increasingly toward adopting for those exact qualities they promise. As things stand today, not making adequate use of sources' power capacity that is available to supply loads in a network is a major cost and challenge for energy companies [4].

There are many different approaches to automating power distribution. It is important to find which one to use for a particular application, because “The study and characterization of distribution system network reconfiguration along with algorithms used for fast and easy operations provide the power distribution engineer several alternatives, where qualitative, application-dependent criteria can be applied following his experience” [5]. In other words, the best automation solution depends on what is important for a particular network. For example, in some contexts, it is important simply to supply as many loads as possible. In one such context, the minimum power flow technique has been proposed to keep as many supply lines active as possible in the power grid of the city of Jaipur, India [6]. In other applications it is radically important to find an optimal network configuration, such as on a navy shipboard power system, where mixed-integer nonlinear programming has been resorted to in order to solve this problem [7]. Mixed-integer nonlinear programming is an NP-hard problem even more complex than the integer linear programming method of optimization used in this thesis [8]!

We aim here to propose an algorithm that is not only very fast, but also places high value on the importance of individual loads over others. This type of solution is useful in a context where the network needs to be fail-safe, where the total capacity may be limited and power needs to be assigned as quickly as possible, such as in our airplane or hospital examples.

1. NETWORK DISCOVERY

When a generator stops working or a line becomes faulty, the topology of the network changes. Pathways between sources and loads may no longer be viable. There is no sure way of knowing what components will remain viable because anything in an electrical circuit can be damaged.

Before a solution can be found for supplying loads with what sources remain, complete knowledge of the network must be gained. In a situation where this is being done manually, this probably involves looking at a switchboard (or breaking out an ammeter in really desperate circumstances). In the scenario where this is automated, it is instead going to be one or more microcontrollers or embedded systems that have to learn the new topology.

The network can really be thought of as existing in two configurations—the old topology, which is the configuration of the network before the fault(s) occurred, and the new topology, which is the configuration of the network after the fault(s) occurred.

For the purpose of this research, we are not going to assume that there are any tricks or shortcuts we can take in order to speed up discovery of the new topology. In a real application, there may be ways to use the old topology to make assumptions about what does or doesn't exist in the new topology. But we want to be able to solve this problem in the general case, and, as mentioned already, anything can be damaged, so there is no sure way of knowing what components will remain viable. So we are going to have to discover the network from *tabula rasa*.

There is a noteworthy advantage to this, which is that it allows for discovery and optimization of the network when other changes have been made besides faults—a new generator may be added to help supply the network, or new lines may be connected together, for example.

In cases where there is a central processor monitoring all components of the network, the discovery phase should be trivial. The most interesting case is the most modular one, where every source, switch, and load point have dedicated microcontrollers monitoring and directing them. Each of these points of interest will be called generically a "node." Every switch node in the network must have a complete picture of the entire network in order to independently find the power assignment solution, and, thereby, whether to close or open the switch it controls.

A network consisting of nodes and links is called a graph. Discovery of all nodes in a graph is a well-known problem to graph theory, and is most simply accomplished by the Bellman-Ford algorithm. (As an aside, if the reader encounters this material elsewhere, "nodes" and "links" are also referred to as "vertices" and "edges" in graph theory.) The Bellman-Ford algorithm is a procedure by which each node learns of its neighbors, as well as the distance to each node in the graph. For our purposes, shortest path calculation is not important, unlike knowing which node is adjacent to which other nodes (among other things), but this could easily be communicated in place of or in addition to distance information. Additionally, the nodes will need to calculate some other information that will be used later, such as which links are incompatible to be used with other links and the order in which the Heuristic Solver should check for links to use when attempting to bridge loads with sources. So the Bellman-Ford algorithm is really a stand-in here for a more sophisticated exchange of packets between the nodes that includes some of this other

information. The purpose is to show how discovery of the network topology could be accomplished.

The Bellman-Ford algorithm to be run on each node could be stated succinctly as follows:

```
1.  self.distances[self] = 0
2.  for link in self.links:
3.      link.send_update()
4.  while True:
5.      for link in self.links:
6.          link.receive_update()
7.      if self.new_distances_recorded:
8.          for link in self.links:
9.              link.send_update()
10.         self.new_distances_recorded = False
```

Pseudocode Block 1.1: Bellman-Ford overview.

This is given as Python pseudocode because most of the code written for this project was written in Python. (C was also used.) Python is a relatively high-level, easy-to-read language, so in order to keep everything consistent, all pseudocode in this thesis will be written in a pythonic manner.

Each node has a class variable called *distances* (line 1). This is a dictionary that contains nothing until the key *self* is given the value 0. (For those not familiar with Python, a dictionary is similar to a hashtable—values are looked up by keys, but both the key and value can be of any variable type. New key-value pairs can be added with a simple assignment statement as in line 1 above, and values can be freely updated.) The *self* that is given as a key for the value 0 is a reference to the node itself. The purpose is to keep track of distances to other nodes in the graph, and because every node knows *a priori* of its own existence and that it is in the exact same location as itself, it can trivially assign the distance to itself as 0 hops away.

Next, in lines 2 and 3, every link in turn is being sent an update with the *send_update* method. These links are simply the lines that are going out of the node in question to other, adjacent nodes. In the beginning, the node does not even necessarily know the identity of the nodes on the other ends of those lines, or indeed even if there are nodes at the other ends of those lines. It just sends packets of information that contains its own *distances* dictionary, so that each of its neighboring nodes can learn what it knows.

Line 4 begins a perpetual process of network discovery, which is not necessary in a graph that is sure to be static or has a known quantity of nodes, but we are operating under the assumption that the topology can change at any time and that the total quantity of nodes is not known by any individual node.

However, in order to solve the power distribution network, the discovery process can obviously not last forever. If the network is never assumed to be fully known, a solution cannot be calculated. We assume the existence of an interval after which, if there have been no discovered changes, discovery can be assumed to be complete for the time being and so a solution can be calculated, after which, changes continue to be scanned for in case a new solution is needed. This interval is to be determined empirically, as this is specific to what the general topology of the power distribution network being discovered is likely to be.

In a given graph, the number of iterations necessary for the Bellman-Ford algorithm to complete is always, at most, one less than the total number of nodes, so if the number of nodes can at least be estimated, this is a point of reference by which to gauge the number of cycles to run before it's assumed that the full topology is known. Additionally, the engineer may want to

add an interval between each scan, to save resources in case it is not important to scan for changes continually (every 50 milliseconds, for example).

Lines 4 and 5 are just to show listening for and processing of packets from incoming links. In the Bellman-Ford algorithm proper, the links directed into the node are not necessarily the same as the links directed out of the node, but because we're just modeling real circuits here, we can assume that every link is bidirectional.

If a packet comes in that contains a *distances* dictionary from another node, then each key-value pair is compared to this node's own *distances* dictionary. There are two cases of interest. First, if the foreign *distances* contains a key that is not already in its own *distances* dictionary, then it records it in its own dictionary with a value at +1 value from the one recorded in the packet received.

For example, let's say Node A's current *distances* dictionary is {'Node A' : 0, 'Node B' : 1, 'Node C' : 1}. It receives a packet that contains the values {'Node B' : 0, 'Node A' : 1, 'Node D' : 1}. Because Node D is not currently in *distances*, it records it with the value given +1. Node A's new *distances* dictionary is {'Node A' : 0, 'Node B' : 1, 'Node C' : 1, 'Node D' : 2}.

Second, if the foreign *distances* contains a key that is already in this node's own *distances* dictionary, but the value is two or more less than the value already recorded in its own *distances*, then it overwrites the value at +1 value from the one recorded in the packet received.

For example, let's say Node A's current *distances* dictionary is {'Node A' : 0, 'Node B' : 1, 'Node C' : 1, 'Node D' : 2}. It receives a packet that contains the values {'Node B' : 2, 'Node C' : 1, 'Node D' : 0}. Because the value 0 for Node D is at least two less than the value 2 that is already

recorded, the value is overwritten at with the value given +1. Node A's new *distances* dictionary is {'Node A' : 0, 'Node B' : 1, 'Node C' : 1, 'Node D' : 1}.

In either of these cases, the node has updated its own *distances* dictionary, which means it has new information that should be passed along to the other nodes in the network. Setting its own *new_distances_recorded* Boolean value to True causes lines 8-10 to be executed next. 8 and 9 are a repeat of lines 2 and 3, which is just there to send out packets with its own *distances* dictionary, and line 10 is just to reset the value of *new_distances_recorded* to False so that this node only sends packets out in subsequent passes if it receives new information about the network.

Let's look at some examples of the Bellman-Ford algorithm in action. One of the simplest possible examples is just a network with one source, one load, and one connecting line with a switch on it. Each of these three objects is a node, and there are two lines connecting them together. Here is what this looks like:

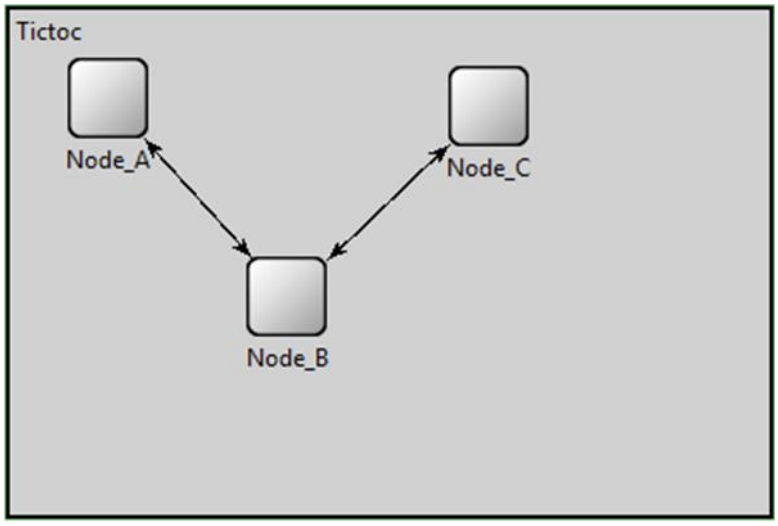

Figure 1.1: Bellman-Ford example 1.

This is an image taken from an Omnet++ simulation (which is why "Tictoc" is seen in the top left corner—the Tictoc example code was modified to create the Bellman-Ford simulation, and the name was never changed). Let's step through it one tic at a time…

	Node A	**Node B**	**Node C**
distances	{'Node A' : 0}	{'Node B' : 0}	{'Node C' : 0}

Table 1.1: Bellman-Ford example 1, *distances* step 0.

To begin with, each node fills its own distances table with what it knows about itself. Then, they must send this information to their neighbors.

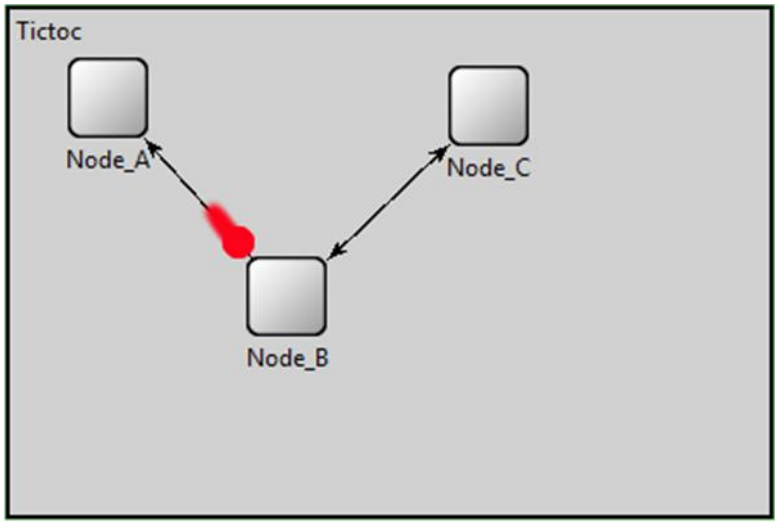

Figure 1.2: Bellman-Ford example 1, step 1, Node A.

Node A sends a packet with its *distances* dictionary along the line to Node B. Looking inside the packet, Node B sees Node A listed with value 0. Node B hasn't heard of Node A before, so it adds it to its own *distances* dictionary with value 1. Node B's *distances* will look like this: {'Node B' : 0, 'Node A' : 1}.

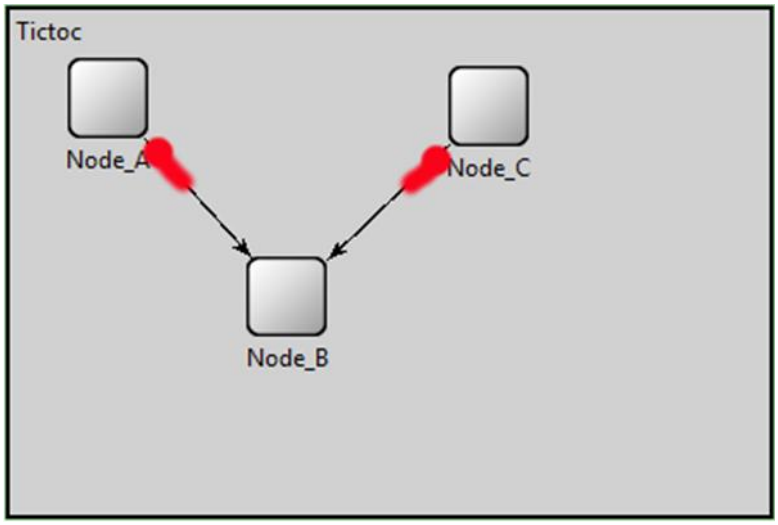

Figure 1.3: Bellman-Ford example 1, step 1, Node B.

Node B sends a packet with its *distances* dictionary along each of its lines, which go to Node A and to Node C. Because this occurs at the exact same time that it's still being sent packets from Node A and Node C, it hasn't updated its *distances* dictionary before sending its own packets. Therefore, the dictionary it's sending out is this: {'Node B' : 0}.

Looking inside the packet it receives, Node A sees Node B listed with value 0. Node A hasn't heard of Node B before, so it adds it to its own *distances* dictionary with value 1. Node A's *distances* will look like this: {'Node A' : 0, 'Node B' : 1}.

Also looking inside the packet that it receives, Node C sees Node B listed with value 0. Node C hasn't heard of Node B before, so it adds it to its own *distances* dictionary with value 1. Node C's *distances* will look like this: {'Node C' : 0, 'Node B' : 1}.

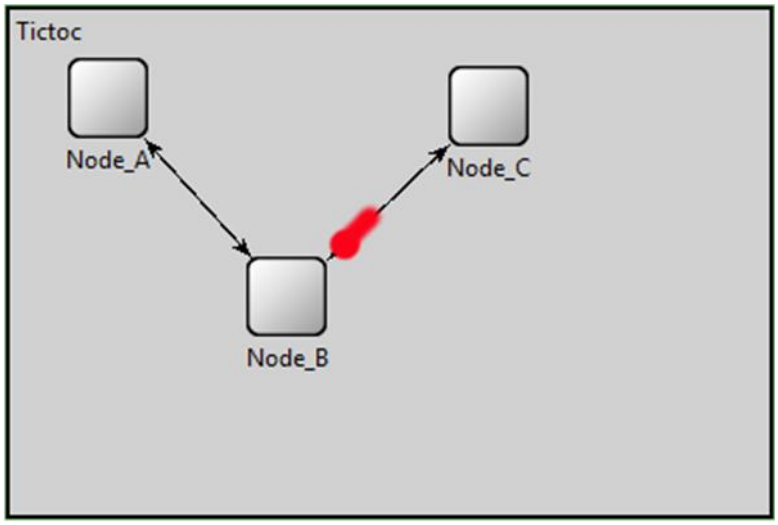

Figure 1.4: Bellman-Ford example 1, step 1, Node C.

Node C sends a packet with its *distances* dictionary along the line to Node B. Again, because all these packets are being sent simultaneously, Node C's *distances* only contains a reference to itself with value 0.

Looking inside this packet it receives, Node B sees Node C listed with value 0. Node B hasn't heard of Node C before, so it adds it to its own *distances* dictionary with value 1. In addition to the information it got from the packet sent by Node A, Node B's *distances* will look like this: {'Node B' : 0, 'Node A' : 1, 'Node C' : 1}.

Here is what each node now knows after one cycle:

	Node A	**Node B**	**Node C**
distances	{'Node A' : 0, 'Node B' : 1}	{'Node B' : 0, 'Node A' : 1, 'Node C' : 1}	{'Node C' : 0, 'Node B' : 1}

Table 1.2: Bellman-Ford example 1, *distances* step 1.

Every node in this graph has received new information about shortest distances to other nodes, so each node is again going to be prompted to send another packet.

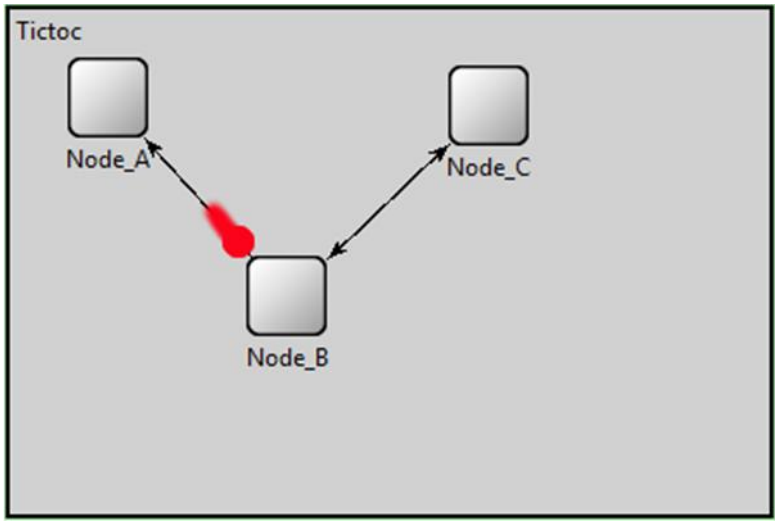

Figure 1.5: Bellman-Ford example 1, step 2, Node A.

Node A sends a packet with its *distances* dictionary along the line to Node B. Looking inside the packet, Node B sees references to Node A at value 0 and Node B at value 1. Node B already knows about both of these nodes, so neither is new. Additionally, the values that Node B has for each of these nodes is at most 1 greater than the values received in the packet, so Node B's *distances* dictionary remains unchanged: {'Node B' : 0, 'Node A' : 1, 'Node C' : 1}.

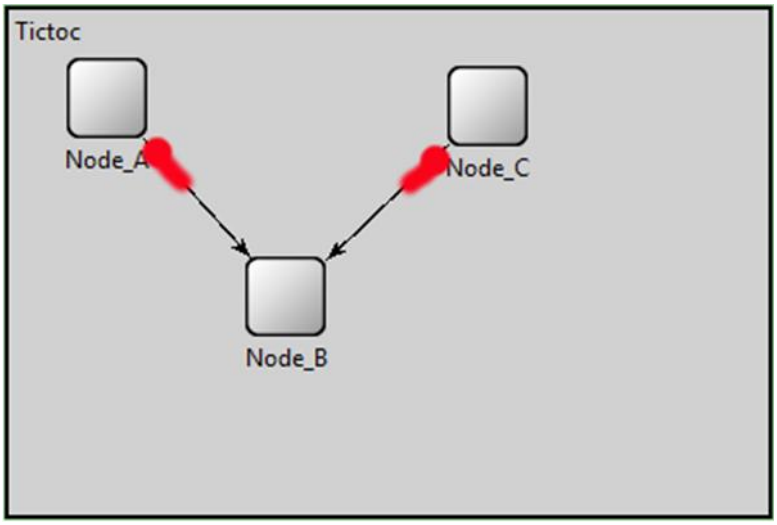

Figure 1.6: Bellman-Ford example 1, step 2, Node B.

Node B sends a packet with its *distances* dictionary along each of its lines, which go to Node A and to Node C. Both Node A and Node C look at the packets they receive, which contains this: {'Node B' : 0, 'Node A' : 1, 'Node C' : 1}. They compare this to their own *distances* dictionaries.

Node A already has a reference to Node A and Node B at values that are at most 1 greater than these values, so it does nothing with those. But Node C is new, so it records it into its own dictionary at +1 value: {'Node A' : 0, 'Node B' : 1, 'Node C': 2}.

Node C already has references to Node C and Node B, and at values that are at most 1 greater than these values listed, so it ignores those. However, Node A is new, so it records it into its own dictionary at +1 value: {'Node C' : 0, 'Node B' : 1, 'Node A' : 2}.

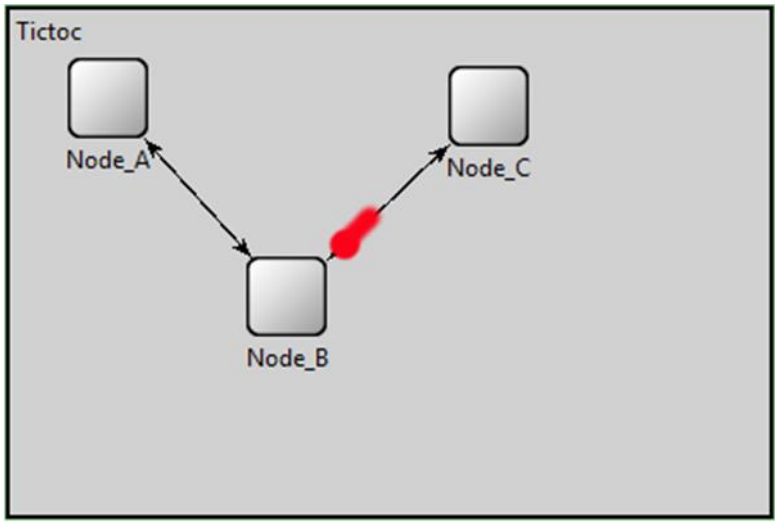

Figure 1.7: Bellman-Ford example 1, step 2, Node C.

Node C sends a packet with its *distances* dictionary along the line to Node B. Again, remember that this occurs simultaneously with being sent the packet from Node B, so it doesn't have the information about Node A from that yet. So the packet sent just contains: {'Node C' : 0, 'Node B' : 1}.

Looking inside the packet, Node B sees that it already knows about both of these nodes. Also, the values that Node B has for each of these nodes is at most 1 greater than the values received in the packet, so Node B's *distances* dictionary remains unchanged: {'Node B' : 0, 'Node A' : 1, 'Node C' : 1}.

Here is what each node now knows after two cycles:

	Node A	**Node B**	**Node C**
distances	{'Node A' : 0, 'Node B' : 1, 'Node C' : 2}	{'Node B' : 0, 'Node A' : 1, 'Node C' : 1}	{'Node C' : 0, 'Node B' : 1, 'Node A' : 2}

Table 1.3: Bellman-Ford example 1, *distances* step 2.

Technically, each node has complete knowledge of the whole network at this point. If it could be guaranteed that the network hasn't changed and the nodes knew how many other nodes there were in the network, they could just stop at this point and move on to finding a power assignment solution.

Since our algorithm makes no such assumption, however, the exchange of packets continues. Node B received no new information this cycle, which means it will be sending no packets this time. However, Node A and Node C each added new information to their *distances* dictionary, so they are each going to send packets.

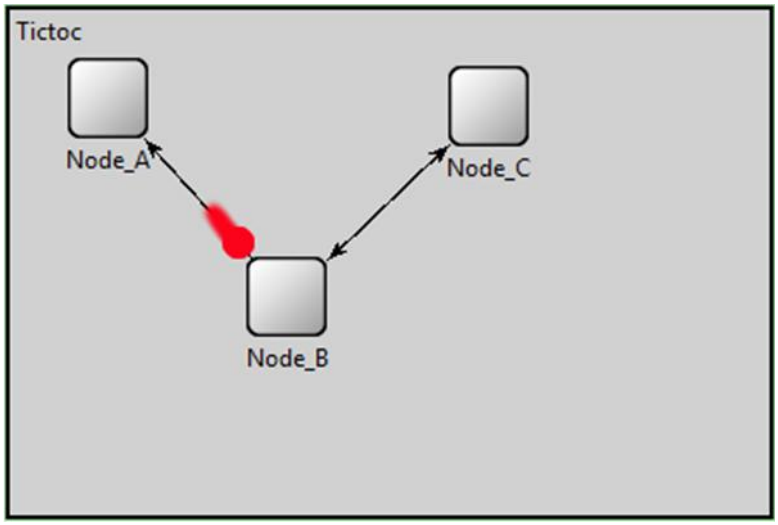

Figure 1.8: Bellman-Ford example 1, step 3, Node A.

Node A sends a packet with its dictionary: {'Node A' : 0, 'Node B' : 1, 'Node C': 2}. Node B already knows about Node A, Node B, and Node C, and the values in its own dictionary are at most 1 more than the values given by this packet, so it doesn't update its own *distances* at all.

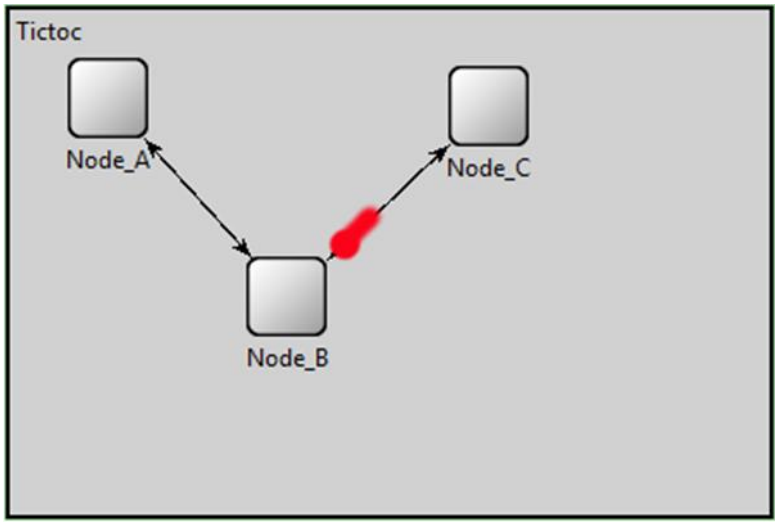

Figure 1.9: Bellman-Ford example 1, step 3, Node C.

Node C sends a packet with its dictionary: {'Node C' : 0, 'Node B' : 1, 'Node A': 2}. Again, Node B already knows about each of the nodes mentioned in this packet, and none of the values are less than 1 below the values in its own *distances*, so it makes no updates.

After three cycles, here is what each node knows about:

	Node A	**Node B**	**Node C**
distances	{'Node A' : 0, 'Node B' : 1, 'Node C' : 2}	{'Node B' : 0, 'Node A' : 1, 'Node C' : 1}	{'Node C' : 0, 'Node B' : 1, 'Node A' : 2}

Table 1.4: Bellman-Ford example 1, *distances* step 3.

And since no *distances* dictionaries have been changed since the last cycle, no nodes are going to send any packets on the fourth cycle. They'll just be listening on the line for new packets. But since no packets are being sent, no packets will be received, so again, no changes will be made. This means no packets will be sent on the fifth cycle, and on and on. At some point, they have to assume that their knowledge of the network is complete and move on to the next stage, finding a solution.

This Bellman-Ford algorithm works for networks of any size. While this has been tested by us on simulations with dozens of nodes, we'll spare the details in order to save dozens of pages. We will, however, look at an example of running this algorithm on a larger network, but with only the illustrations of packet exchanges and table updates. To save space, packet exchanges that take place simultaneously will be shown happening at the same time on the same graphic.

Here is the initial setup:

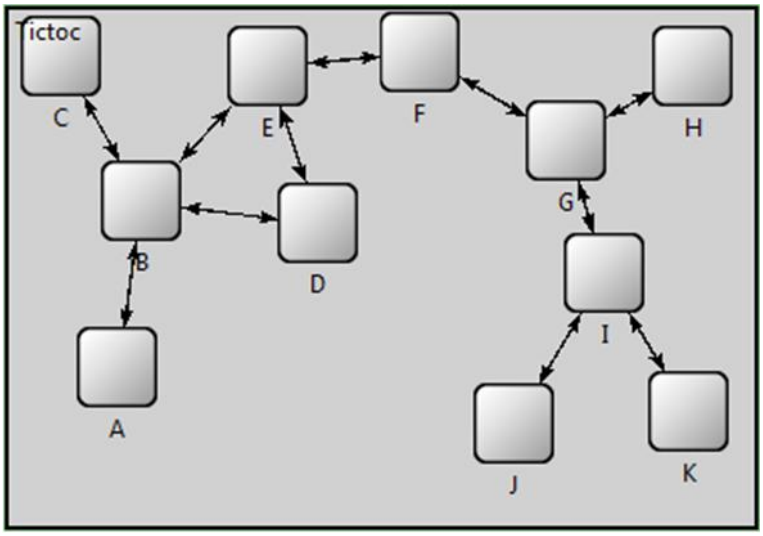

Figure 1.10: Bellman-Ford example 2.

Here is the initial table:

Node:	A	B	C	D	E	F	G	H	I	J	K
distances	{'A':0}	{'B':0}	{'C':0}	{'D':0}	{'E':0}	{'F':0}	{'G':0}	{'H':0}	{'I':0}	{'J':0}	{'K':0}

Table 1.5: Bellman-Ford example 2, *distances* step 0.

Messages sent on the first cycle:

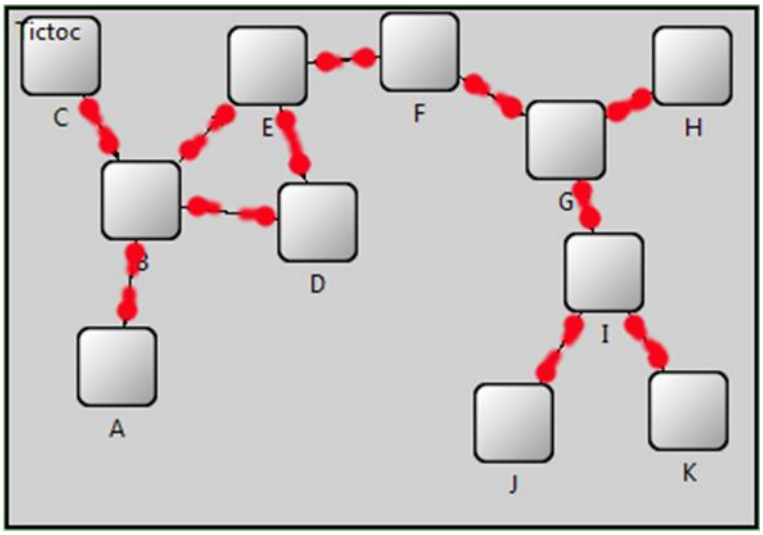

Figure 1.11: Bellman-Ford example 2, step 1.

Table after first cycle:

Node:	A	B	C	D	E	F	G	H	I	J	K
distances	{'A':0, 'B':1}	{'B':0, 'A':1, 'C':1, 'D':1, 'E':1}	{'C':0, 'B':1}	{'D':0, 'B':1, 'E':1}	{'E':0, 'B':1, 'D':1, 'F':1}	{'F':0, 'E':1, 'G':1}	{'G':0, 'F':1, 'H':1, 'I':1}	{'H':0, 'G':1}	{'I':0, 'G':1, 'J':1, 'K':1}	{'J':0, 'I':1}	{'K':0, 'I':1}

Table 1.6: Bellman-Ford example 2, *distances* step 1.

Messages sent on second cycle:

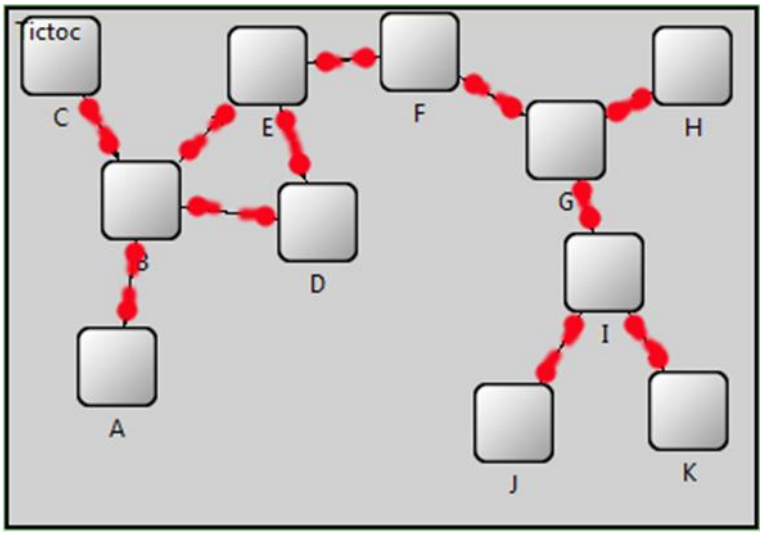

Figure 1.12: Bellman-Ford example 2, step 2.

Table after second cycle:

Node:	A	B	C	D	E	F	G	H	I	J	K
distances	{'A':0, 'B':1, 'C':2, 'E':2, 'D':2}	{'B':0, 'A':1, 'C':1, 'D':1, 'E':1, 'F':2}	{'C':0, 'B':1, 'A':2, 'E':2, 'D':2}	{'D':0, 'B':1, 'E':1, 'A':2, 'C':2, 'F':2}	{'E':0, 'B':1, 'D':1, 'F':1, 'A':2, 'C':2, 'G':2}	{'F':0, 'E':1, 'G':1, 'B':2, 'H':2, 'I':2, 'D':2}	{'G':0, 'F':1, 'H':1, 'I':1, 'E':2, 'J':2, 'K':2}	{'H':0, 'G':1, 'F':2, 'I':2}	{'I':0, 'G':1, 'J':1, 'K':1, 'F':2, 'H':2}	{'J':0, 'I':1, 'G':2, 'K':2}	{'K':0, 'I':1, 'G':2, 'J':2}

Table 1.7: Bellman-Ford example 2, *distances* step 2.

Messages sent on third cycle:

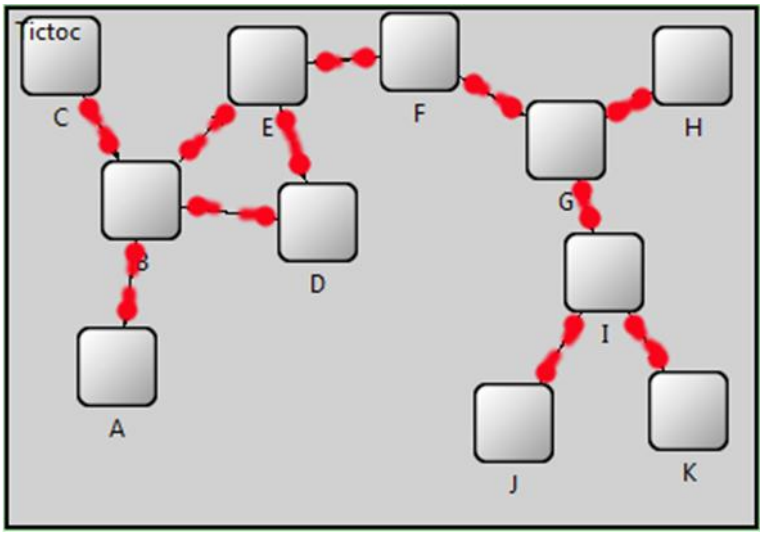

Figure 1.13: Bellman-Ford example 2, step 3.

Table after third cycle:

Node:	A	B	C	D	E	F	G	H	I	J	K
distances	{'A':0, 'B':1, 'C':2, 'E':2, 'D':2, 'F':3}	{'B':0, 'A':1, 'C':1, 'D':1, 'E':1, 'F':2, 'G':3}	{'C':0, 'B':1, 'A':2, 'E':2, 'D':2, 'F':3}	{'D':0, 'B':1, 'E':1, 'A':2, 'C':2, 'F':2, 'G':3}	{'E':0, 'B':1, 'D':1, 'F':1, 'A':2, 'C':2, 'G':2, 'H':3, 'I':3}	{'F':0, 'E':1, 'G':1, 'B':2, 'H':2, 'I':2, 'D':2, 'A':3, 'C':3, 'J':3, 'K':3}	{'G':0, 'F':1, 'H':1, 'I':1, 'E':2, 'J':2, 'K':2, 'B':3, 'D':3}	{'H':0, 'G':1, 'F':2, 'I':2, 'E':3, 'J':3, 'K':3}	{'I':0, 'G':1, 'J':1, 'K':1, 'F':2, 'H':2, 'E':3}	{'J':0, 'I':1, 'G':2, 'K':2, 'F':3, 'H':3}	{'K':0, 'I':1, 'G':2, 'J':2, 'F':3, 'H':3}

Table 1.8: Bellman-Ford example 2, *distances* step 3.

Messages sent on fourth cycle:

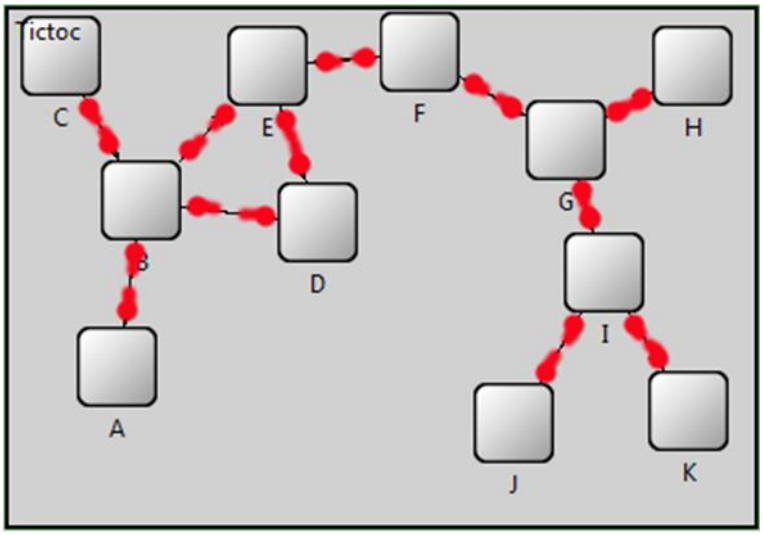

Figure 1.14: Bellman-Ford example 2, step 4.

Table after fourth cycle:

Node:	A	B	C	D	E	F	G	H	I	J	K
distances	{'A':0, 'B':1, 'C':2, 'E':2, 'D':2, 'F':3, 'G':4}	{'B':0, 'A':1, 'C':1, 'D':1, 'E':1, 'F':2, 'G':3, 'H':4, 'I':4}	{'C':0, 'B':1, 'A':2, 'E':2, 'D':2, 'F':3, 'G':4}	{'D':0, 'B':1, 'E':1, 'A':2, 'C':2, 'F':2, 'G':3, 'H':4, 'I':4}	{'E':0, 'B':1, 'D':1, 'F':1, 'A':2, 'C':2, 'G':2, 'H':3, 'I':3, 'J':4, 'K':4}	{'F':0, 'E':1, 'G':1, 'B':2, 'H':2, 'I':2, 'D':2, 'A':3, 'C':3, 'J':3, 'K':3}	{'G':0, 'F':1, 'H':1, 'I':1, 'E':2, 'J':2, 'K':2, 'B':3, 'D':3, 'A':4, 'C'4}	{'H':0, 'G':1, 'F':2, 'I':2, 'E':3, 'J':3, 'K':3, 'B':4, 'D':4}	{'I':0, 'G':1, 'J':1, 'K':1, 'F':2, 'H':2, 'E':3, 'B':4, 'D':4}	{'J':0, 'I':1, 'G':2, 'K':2, 'F':3, 'H':3, 'E':4}	{'K':0, 'I':1, 'G':2, 'J':2, 'F':3, 'H':3, 'E':4}

Table 1.9: Bellman-Ford example 2, *distances* step 4.

Messages sent on fifth cycle:

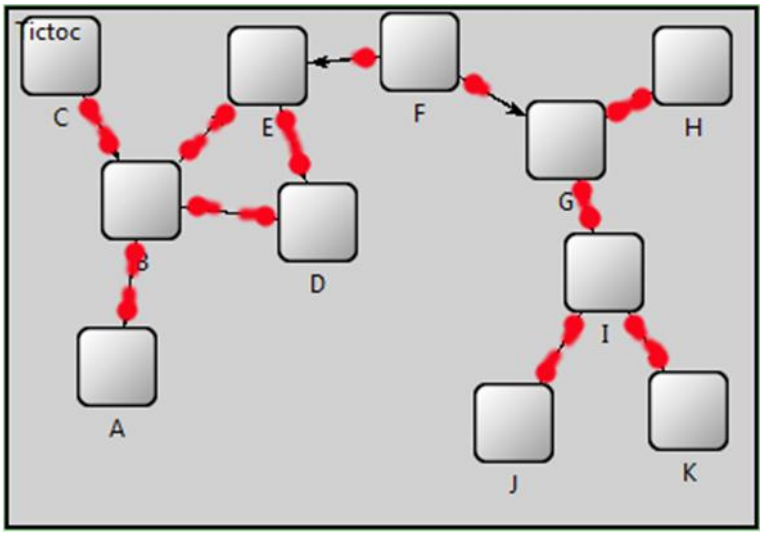

Figure 1.15: Bellman-Ford example 2, step 5.

Table after fifth cycle:

Node:	A	B	C	D	E	F	G	H	I	J	K
distances	{'A':0, 'B':1, 'C':2, 'E':2, 'D':2, 'F':3, 'G':4, 'H':5, 'I':5}	{'B':0, 'A':1, 'C':1, 'D':1, 'E':1, 'F':2, 'G':3, 'H':4, 'I':4, 'J':5, 'K':5}	{'C':0, 'B':1, 'A':2, 'E':2, 'D':2, 'F':3, 'G':4, 'H':5, 'I':5}	{'D':0, 'B':1, 'E':1, 'A':2, 'C':2, 'F':2, 'G':3, 'H':4, 'I':4, 'J':5, 'K':5}	{'E':0, 'B':1, 'D':1, 'F':1, 'A':2, 'C':2, 'G':2, 'H':3, 'I':3, 'J':4, 'K':4}	{'F':0, 'E':1, 'G':1, 'B':2, 'H':2, 'I':2, 'D':2, 'A':3, 'C':3, 'J':3, 'K':3}	{'G':0, 'F':1, 'H':1, 'I':1, 'E':2, 'J':2, 'K':2, 'B':3, 'D':3, 'A':4, 'C'4}	{'H':0, 'G':1, 'F':2, 'I':2, 'E':3, 'J':3, 'K':3, 'B':4, 'D':4, 'A':5, 'C':5}	{'I':0, 'G':1, 'J':1, 'K':1, 'F':2, 'H':2, 'E':3, 'B':4, 'D':4, 'A':5, 'C':5}	{'J':0, 'I':1, 'G':2, 'K':2, 'F':3, 'H':3, 'E':4, 'B':5, 'D':5}	{'K':0, 'I':1, 'G':2, 'J':2, 'F':3, 'H':3, 'E':4, 'B':5, 'D':5}

Table 1.10: Bellman-Ford example 2, *distances* step 5.

Messages sent on sixth cycle:

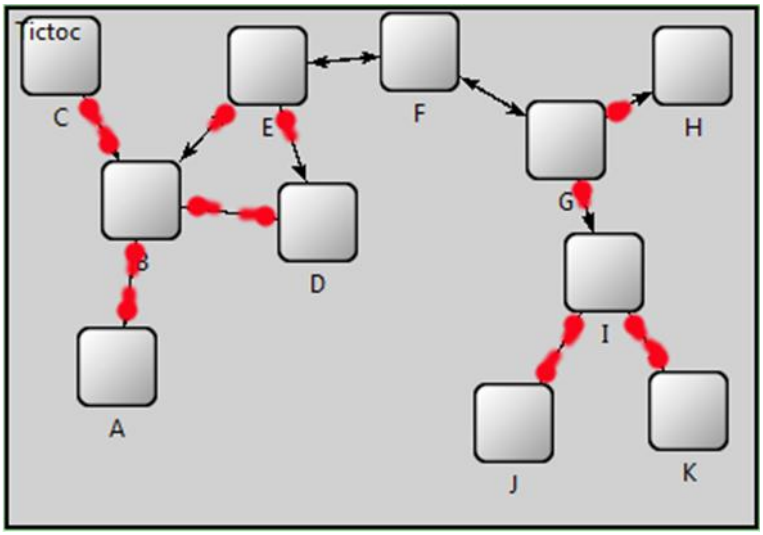

Figure 1.16: Bellman-Ford example 2, step 6.

Table after sixth cycle:

Node:	A	B	C	D	E	F	G	H	I	J	K
distances	{'A':0, 'B':1, 'C':2, 'E':2, 'D':2, 'F':3, 'G':4, 'H':5, 'I':5, 'J':6, 'K':6}	{'B':0, 'A':1, 'C':1, 'D':1, 'E':1, 'F':2, 'G':3, 'H':4, 'I':4, 'J':5, 'K':5}	{'C':0, 'B':1, 'A':2, 'E':2, 'D':2, 'F':3, 'G':4, 'H':5, 'I':5, 'J':6, 'K':6}	{'D':0, 'B':1, 'E':1, 'A':2, 'C':2, 'F':2, 'G':3, 'H':4, 'I':4, 'J':5, 'K':5}	{'E':0, 'B':1, 'D':1, 'F':1, 'A':2, 'C':2, 'G':2, 'H':3, 'I':3, 'J':4, 'K':4}	{'F':0, 'E':1, 'G':1, 'B':2, 'H':2, 'I':2, 'D':2, 'A':3, 'C':3, 'J':3, 'K':3}	{'G':0, 'F':1, 'H':1, 'I':1, 'E':2, 'J':2, 'K':2, 'B':3, 'D':3, 'A':4, 'C'4}	{'H':0, 'G':1, 'F':2, 'I':2, 'E':3, 'J':3, 'K':3, 'B':4, 'D':4, 'A':5, 'C':5}	{'I':0, 'G':1, 'J':1, 'K':1, 'F':2, 'H':2, 'E':3, 'B':4, 'D':4, 'A':5, 'C':5}	{'J':0, 'I':1, 'G':2, 'K':2, 'F':3, 'H':3, 'E':4, 'B':5, 'D':5, 'A':6, 'C':6}	{'K':0, 'I':1, 'G':2, 'J':2, 'F':3, 'H':3, 'E':4, 'B':5, 'D':5, 'A':6, 'C':6}

Table 1.11: Bellman-Ford example 2, *distances* step 6.

Note that at this point, there each node has complete information about each other node in the graph. They will continue running in perpetuity, but they will discover no new nodes and learn of no shorter paths than the ones they already know about. For this reason, this table will not be shown again, but the final message exchange in cycle seven will be shown, before they cease entirely from cycle eight and upward.

Messages sent on seventh cycle:

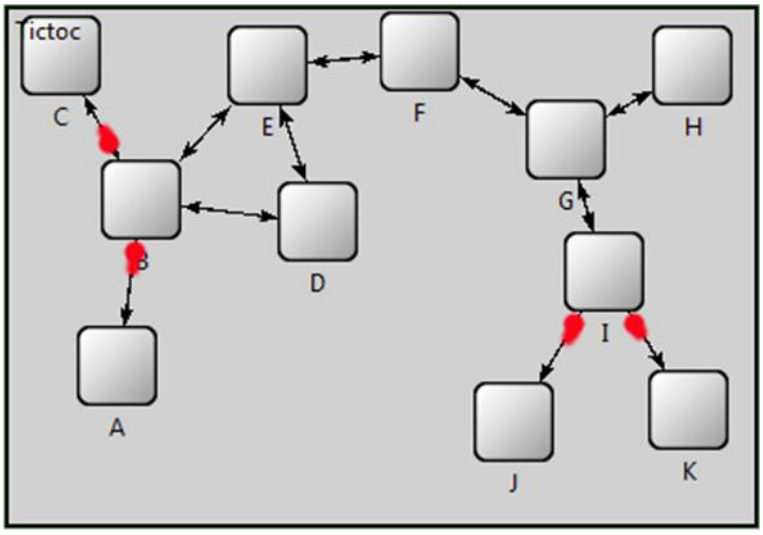

Figure 1.17: Bellman-Ford example 2, step 7.

It should be noted that the examples given here operate under the assumption that each node is independently sending and receiving messages at the same time. This is fine for a simulation, but in a real distributed communication network, it may not be practical to rely on such synchronicity. It is good, then, that the Bellman-Ford algorithm can run asynchronously, without ruining the results [9]. Each node must simply execute another loop of the algorithm after a given time interval.

One limitation of the Bellman-Ford algorithm is that it is optimistic, meaning that if the topology of the network changes and the distance between one node and another grows longer, or if a node is taken out of the network, it does not have a way to account for this. Each node believes that all the nodes it's heard about still exist, and it still thinks that the distance to them are the smallest number of hops that it's calculated it takes to get there. To account for this optimism, after the same interval that each node is assuming it has a complete picture of the

network, it can restart the algorithm with a new *distances* table in order to confirm that the topology is as it believes it to be. If there have been any changes to the network since the last interval, then a new solution for power assignment should be calculated. Otherwise, the solution should remain as is.

Of course, changes in distance don't matter to the power assignment algorithms except inasmuch as they reflect a change in network topology, but distance is just a stand-in here for other sorts of information to be exchanged—namely, which nodes are connected to which other nodes, which nodes are sources and what are their capacities, and which nodes are loads and what are their priorities and load requirements.

2. POWER RECONFIGURATION ALGORITHMS

We present a fast, efficient method for power distribution network reconfiguration. This is a heuristic algorithm for assigning sources to supply as many loads with power as possible, favoring loads of higher priority over loads of lower priority. This algorithm will be referred to as the Heuristic Solver. To judge the quality of solutions that the Heuristic Solver produces, it will be compared to solutions produced by an integer linear programming formulation. The algorithm used to formulate the network as an integer linear programming problem will be referred to as the ILP Solver. Linear programming is a popular tool for optimization in the field of operations research used to optimize the desired outcome in a mathematical model. The solutions produced by it must necessarily be optimal. However, integer linear programming is an NP hard problem, so we will measure the feasibility of relying on the ILP Solver for power assignment solutions to networks of varying complexity. The Heuristic Solver primarily uses a method called augmented path matching, which, like the Bellman-Ford algorithm, comes from graph theory.

In order to describe these algorithms more easily, we're going to use simple units. We assume that capacities and priority levels are given as whole numbers. Load requirements have unit values. Each of these solvers could just as easily handle floating point values for any of these variables, but we limit them to small integer values for the purpose of demonstrating simply the effectiveness of these algorithms relative to each other. We don't refer to units such as volts or amperes either—they are abstract units.

One other important detail to mention about the results is that these are good numbers for making relative comparisons between the two algorithms, not necessarily for seeing how they

are going to perform individually on whatever device it will be run on. There are two reasons for this: one, the language, and two, the hardware.

Python 2.7 was used for the development and testing of these solvers, which is a very high-level language written in C. Python is great for rapid development and testing, given that it is easy to make changes with and it is an interpreted language, which means that the source code is compiled by the Python interpreter at the point of execution. The downside to these conveniences is that it is a comparatively slow language. Depending on the task, taking sometimes 10 or 20 times longer to execute than a similar program written in C, which should run approximately as fast as assembly.

The hardware used to run all these tests is a laptop computer with a 2 GHz Intel Core2 Duo processor, 6 GB of RAM, and which runs the 64-bit version of Windows 7 Enterprise Edition (Service Pack 1). If this code needed to run on a server, or incorporate network programming to mimic running it on an embedded system, we would have probably chosen to write it for Linux. These test programs could still yet be ported to other operating systems, but as of now it's using some libraries that are Windows-specific because it was more important to make a sort of interactive toy that could be widely distributed for others to test as well.

The given limitations of the language and hardware that are being used don't matter all that much. There's no need to super-optimize these solvers for the sake of testing, because it is not necessarily the case that they will be utilized in a program that is written in C or a low-level language. The target program may be written in Java, running on an old desktop in a power station that's running Windows XP. There are many kinds of other, more esoteric things one

might run into in the tech industry. So, running it with Python on an old laptop should be sufficient to compare these solvers. Especially because they both face the same restraints of having to use this sluggish selection of language, OS, and hardware. Perhaps the results that the Heuristic Solver return will impress more, given these limitations?

But first, we're going to look at the ILP Solver. Integer linear programming is a type of linear programming. Linear programming is solving a system of equations that have a linear relationship in order to find the optimal value (minimum or maximum) for an objective function. Constraints are added to the system to bound variables within specified ranges. Integer linear programming just adds the stipulation that the variables must also be integers.

The solving of linear programming problems has been relegated to the Python module *PuLP*, which utilizes the COIN-OR CBC solver. There is a good reference in the Citation section that gives the general idea of how linear programming works and what it is used for [10]. For our purposes, it seems the best way to demonstrate how we make use of it in the ILP Solver is to use an example. Here is a flattened view of a network that needs to be solved:

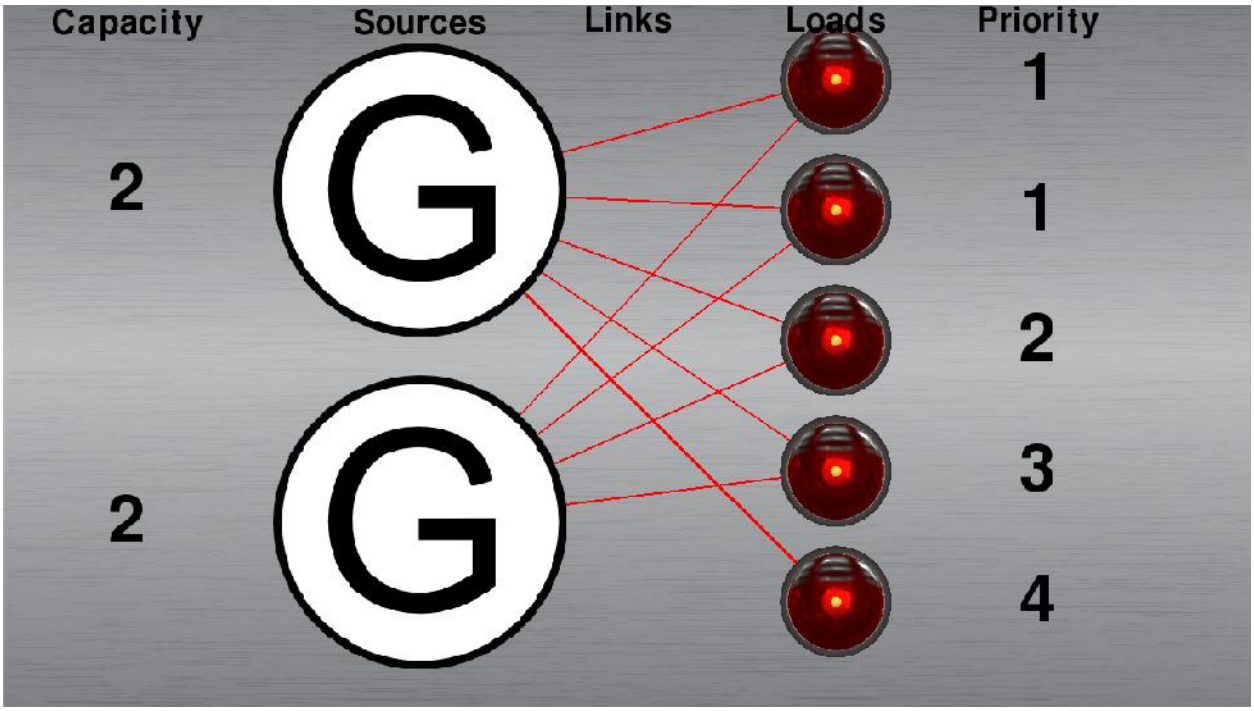

Figure 2.1: Network example.

Ordinarily, the first step is to consider any active links and loads to be inactive, but there are no active loads or links in this initial setup. As the diagram shows, there are two sources that supply five loads. Each of the sources have a capacity of 2, which means that they can each supply up to two loads with power. The loads have priorities of 1, 1, 2, 3, and 4, with 1 being the lowest rank in importance and 4 being the highest. Obviously, with two generators each able to supply at most 2 loads, there is a scarcity of supply. In the best case, we can hope for 4 out of 5 loads to be matched with a generator, with one of the priority 1 loads being left out.

Let's look again at this diagram, but with labels included so we can make our linear programming formulation:

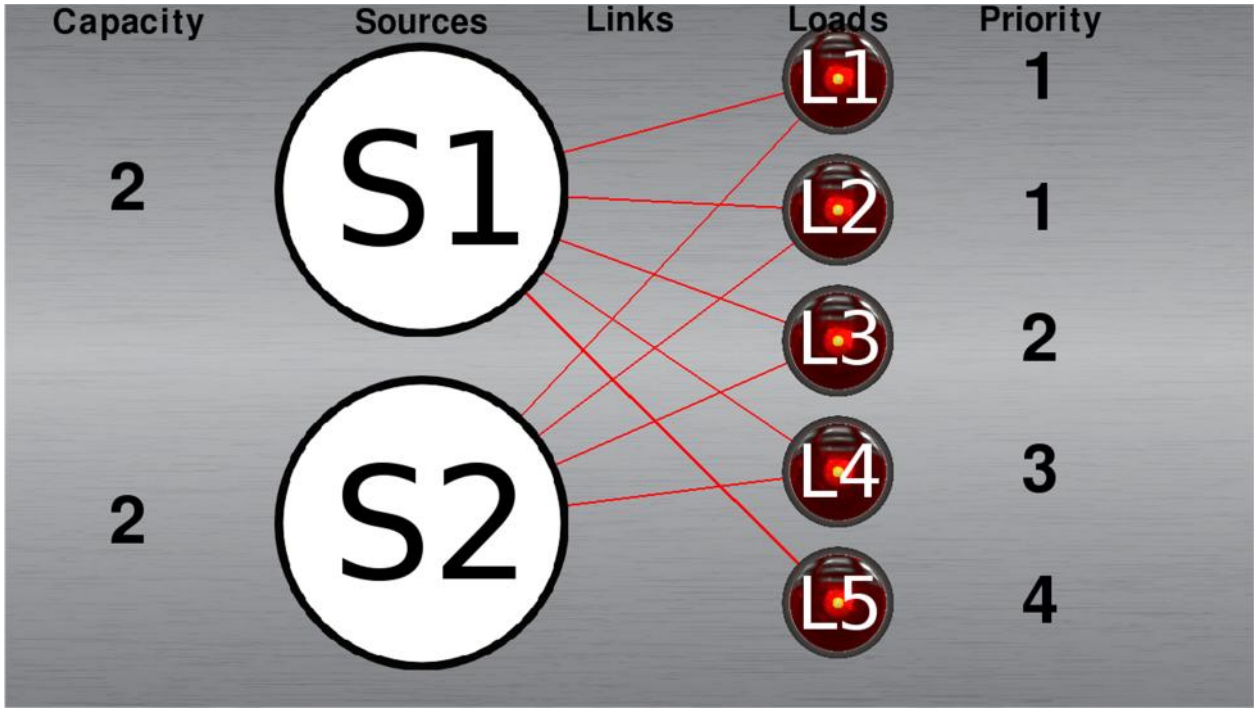

Figure 2.2: Network example, labeled.

These are the names of the links and their incompatibilities with other links in the network:

Links	Incompatibilities
1->1_1	2->4_1
1->2_1	
1->3_1	2->2_1
1->4_1	
1->5_1	
1->5_2	
2->1_1	
2->2_1	1->3_1
2->3_1	2->4_1
2->4_1	2->3_1; 1->1_1

Table 2.1: Example network incompatibilities.

The way these links are named is “S->L_N,” where S is the numeric identifier of the source, L is the numeric identifier of the load, and N is the index of link between that particular source

and load pair. It is extremely difficult to see in the diagram, but there are two links between S1 and L5. Where more than one link exists, they are displayed on top of each other, and the ones below are one pixel thicker on each side. It is displayed like this because there may be many links between the same source and load, and this is the best way to fit them all in the same graphic.

Incompatibilities between links are just that—two or more links that cannot be activated at the same time as one another. So, for example, if 1->1_1 is to be turned on, then the final solution cannot also contain the link 2->4_1. Where a link is listed as incompatible with another, that other link is also listed as incompatible with it. Incompatibilities are also the reason to include the possibility of multiple links between the same source and load—if a source cannot supply a load through one link because of its incompatibilities with other links, then perhaps it can supply it by another link that to the same load that has a different set of incompatibilities. More about incompatibilities and how they occur is in the next section, Flattening the Distribution Network. For now, we just assume the existence of some arbitrary incompatibilities in the example network.

There is some nuance to the priority property of these loads. It is not enough to simply rank them in order of importance. There must also be a decision about what those priority levels mean—are two priority level 1 loads equal to one priority level 2 load? What about two priority level 2 loads against one priority level 3 load? Are they worth a number of points equal to their priority value or does it take two of one level to match the one above it? (Would it take three or would it take four level 1 loads to equal a level 3 load?)

To be as realistic as possible, the decision about how to interpret priority levels is this—each priority level unequivocally outclasses all the priority levels below it. No amount of level 1 loads will outmatch a single level 2 load. This is because when loads are ranked by a certain level of importance, we assume it is for good reason. It doesn't matter how many coffee makers can be supplied on the aircraft if the lights go out, because then no one can see to use them. It doesn't matter how many coffee makers or lights can be supplied if the navigation system goes out.

Therefore, the simple, linear progression of priority levels from one level to another must be weighted such that a single load of each level is worth more than all the priorities below it. There is a recursive formula for this:

```
1.  def priority_weights(priorities):
2.      priority_set = set(priorities)
3.      priority_analogues = dict([(priority, None) for priority in priority_set])
4.      priority_level = 1
5.      weight_snowball = 0
6.      for priority in sorted(priority_set):
7.          priority_analogues[priority] = priority_level
8.          weight_snowball += priority_level * priorities.count(priority)
9.          priority_level = weight_snowball + 1
10.     return priority_analogues
```

Pseudocode Block 2.1: *priority_weights* function.

This is a function that takes a list of given load priorities and returns a dictionary that contains their weighted analogues. The best way to walk through this code is with an example, so let's use the priorities of the loads in our example network: [1, 1, 2, 3, 4].

In line 2, a set is created from this list of priorities. A set object does not contain duplicates of any objects, so the values in *priority_set* are (1, 2, 3, 4).

Line 3 creates a dictionary to be filled with the values for priority weights—it's what we're trying to find and return with this function. Right now, this dictionary only contains the keys that are in *priority_set*: {1 : None, 2 : None, 3 : None, 4 : None}.

Lines 4 and 5 are creating variables that will be incremented to increase the priority weights as we iterate through the set.

The main loop begins on line 6. The *sorted* function returns a list of values in the given input ordered from least to greatest. The reason this is needed is because sets are necessarily unordered collections of unique objects. We also can't just use *priorities*, because we don't want to count the same priority level twice.

On each pass through the loop, the given priority is going to be assigned an analogous weighted value, which is recorded into *priority_analogues*. Let's walk through that process with the priorities we've given…

priority == 1: In line 7, *priority_analogues*[1] is given a value of 1, which is the current value of *priority_level*. *priority_analogues* is now {1 : 1, 2 : None, 3 : None, 4 : None}. Next, in line 8, *weight_snowball* is incremented by 2, which is the current *priority_level* multiplied by the number of times that this *priority* appears in the list *priorities* (1 * 2). The value of *weight_snowball* is now 2. Then, in line 9, *priority_level* is assigned a value of 3, which is the current *weight_snowball* plus 1 (2 + 1).

priority == 2: In line 7, *priority_analogues*[2] is given a value of 3, which is the current value of *priority_level*. *priority_analogues* is now {1 : 1, 2 : 3, 3 : None, 4 : None}. Next, in line 8, *weight_snowball* is incremented by 3, which is the current *priority_level* multiplied by the

number of times that this *priority* appears in the list *priorities* (3 * 1). The value of *weight_snowball* is now 5. Then, in line 9, *priority_level* is assigned a value of 6, which is the current *weight_snowball* plus 1 (5 + 1).

priority == 3: In line 7, *priority_analogues*[3] is given a value of 6, which is the current value of *priority_level*. *priority_analogues* is now {1 : 1, 2 : 3, 3 : 6, 4 : None}. Next, in line 8, *weight_snowball* is incremented by 6, which is the current *priority_level* multiplied by the number of times that this *priority* appears in the list *priorities* (6 * 1). The value of *weight_snowball* is now 11. Then, in line 9, *priority_level* is assigned a value of 12, which is the current *weight_snowball* plus 1 (11 + 1).

priority == 4: In line 7, *priority_analogues*[4] is given a value of 12, which is the current value of *priority_level*. *priority_analogues* is now {1 : 1, 2 : 3, 3 : 6, 4 : 12}. The dictionary *priority_analogues* is now completely filled in, and can be returned in line 10. Lines 8 and 9 still execute, wherein *weight_snowball* is incremented and *priority_level* is assigned a greater value, but it doesn't matter because those variables aren't used again in this function.

For our network, now that we have the *priorities_analogue* for our priorities, we can fill in the weighted priority values:

Load	**Priority**	**Weighted Priority**
L1	1	1
L2	1	1
L3	2	3
L4	3	6
L5	4	12

Table 2.2: Example network weighted priorities.

Notice that no combination of lower-level weighted priorities will equal a priority weight of a higher level. Some examples:

$$1 + 1 == 2 <= 3$$

$$1 + 3 == 4 <= 6$$

$$1 + 1 + 3 + 6 == 11 <= 12$$

Now we can formulate the linear programming problem. This is done programmatically in the general case, but let's walk through this process with our network to better understand it. The first thing to do is define our variables—how is this network represented mathematically? Given certain constraints, only two sets of variables are necessary: one for every link, and one for every load. These integer variables will be constrained to a lower bound of 0, representing "off" or "inactive," and 1, representing "on" or "active." An "x_" will be prepended to each full link name to represent these link variables, and an "l_" will be prepended to each load identifier to represent these load variables. They are as follows:

$$0 <= x\_1->1\_1, x\_1->2\_1, x\_1->3\_1, x\_1->4\_1,$$
$$x\_1->5\_1, x\_1->5\_2, x\_2->1\_1, x\_2->2\_1,$$
$$x\_2->3\_1, x\_2->4\_1, l\_1, l\_2, l\_3, l\_4, l\_5 <= 1$$

Next, we must set the objective function, which is the function to be minimized or maximized. In this case, we want to maximize the total weighted priorities of loads that get assigned power, so our objective function reads:

$$\text{Maximize } (l\_1 * 1) + (l\_2 * 1) + (l\_3 * 3) + (l\_4 * 6) + (l\_5 * 12)$$

Now, in order to show that each load can only be supplied by one link attached to it, we add the following constraints:

(x_1->1_1 + x_2->1_1) == l_1

(x_1->2_1 + x_2->2_1) == l_2

(x_1->3_1 + x_2->3_1) == l_3

(x_1->4_1 + x_2->4_1) == l_4

(x_1->5_1 + x_1->5_2) == l_5

After this, we need to add constraints that enforce rules for source capacity. The number of links going out of each source that can be active is only as many as that source's capacity allows. This is represented thusly:

(x_1->1_1 + x_1->2_1 + x_1->3_1 + x_1->4_1 + x_1->5_1 + x_1->5_2) <= 2

(x_2->1_1 + x_2->2_1 + x_2->3_1 + x_2->4_1) <= 2

(It should be noted that if we were going to allow each load to require other than unit capacity, each load variable individually would be multiplied its load requirement in the constraints listed here.)

Finally, each link cannot be activated at the same time that a link that's incompatible with it. So for each pair of incompatibilities in the network, only as many as one of them can be turned on. These are the constraints for our network that accomplish this:

1->1_1 + 2->4_1 <= 1

1->3_1 + 2->2_1 <= 1

2->3_1 + 2->4_1 <= 1

With our whole network formulated as an integer linear programming problem, the solver can find the solution for us. Here is a diagram of this network as solved by the ILP Solver:

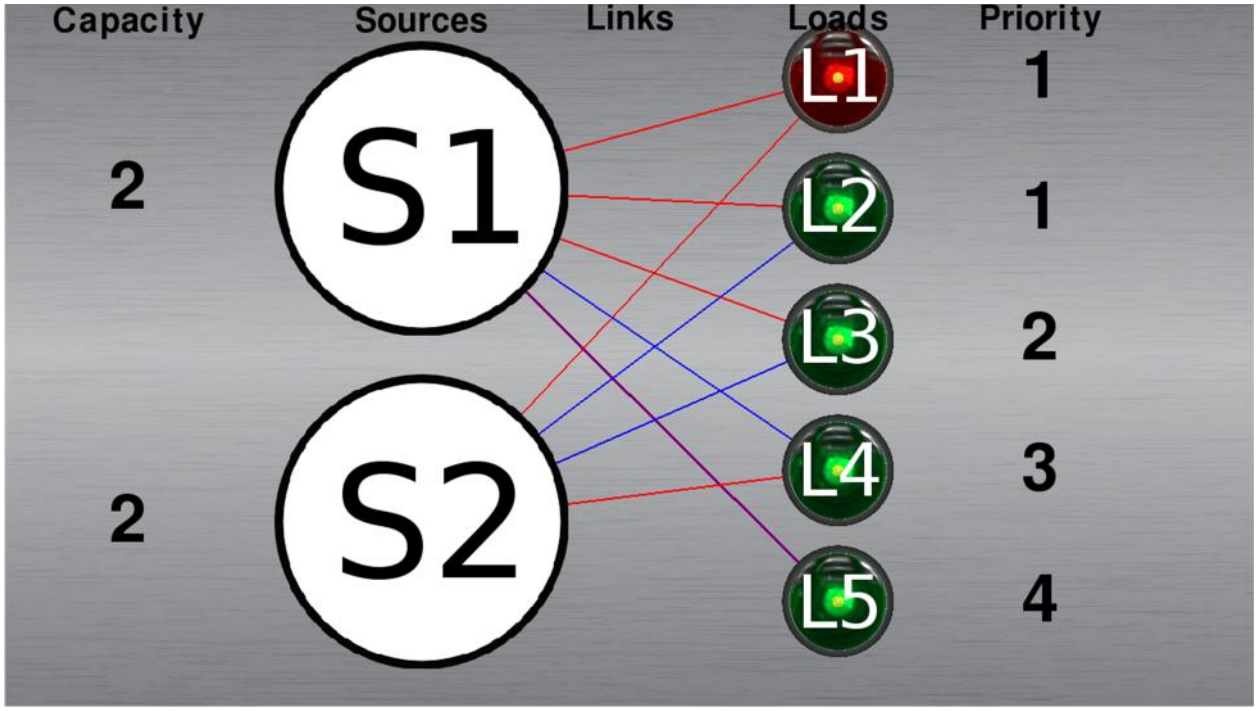

Figure 2.3: Example network ILP Solution.

Red links and loads are inactive. Blue lines and green loads are active. It might look like there is a purple line going from S1 to L5, but that is actually two links—an active one on top of an inactive one. In tabulated form, here are the active links:

Links	Active
1->1_1	False
1->2_1	False
1->3_1	False
1->4_1	**True**
1->5_1	**True**
1->5_2	False
2->1_1	False
2->2_1	**True**
2->3_1	**True**

2->4_1	False

Table 2.3: Example network active links, ILP Solution.

How do we measure the success of a solution? How many loads were assigned is one measure for sure, but that is not as important as the priorities of the loads that were assigned. As stated earlier, it could be that our solution provides power to 10 coffeemakers, but that is still not as good as a solution that just provides power to the navigation system. Therefore, a solution should be measured by the total number of priorities that get assigned in that solution. But not just the total of priorities added linearly—again, two priority level 1 loads do not equal one priority level 2 load. It should instead be measured by the sum total of weighted priorities, which we have already calculated, for the loads that have been assigned. This is called a solution's "composite score." The composite score for the ILP Solution for this network is the total of all the assigned loads' weighted priorities (1 + 3 + 6 + 12), which is 22. The composite score can only be a metric of proportion—how does this solution score against the ideal? This is because each network could have a different total possible composite score, depending on how many loads there are in the network and what their priority levels are.

The composite score measures the quality of a solution for sure, but another important metric is how long it took for the solver to find a solution. On this machine, it took 1.559 seconds for the ILP Solver to find one for this example network. The ILP Solution is guaranteed to be the best solution, but 1.559 seconds is a bit ponderous for such a simple network.

Can we do better? Let's see how the Heuristic Solver compares. Much like with the ILP Solver, the first step in the Heuristic Solver is to consider all loads and links to be initially deactivated. This step is to find a new solution with all possible assignments available. Again, the

network we're using as an example doesn't have anything activated in its initial setup, but for the general case it is important to mention all the steps.

Next, the Heuristic Solver follows this sequence:

```
1.  currently_active_links = set()
2.  orphan_load_stack = loads
3.  by_priority = lambda load: load.priority
4.  orphan_load_stack.sort(key=by_priority, reverse=True)
5.  previous_orphan_count = len(orphan_load_stack) + 1
6.  while (orphan_load_stack) and (len(orphan_load_stack) < previous_orphan_count):
7.      previous_orphan_count = len(orphan_load_stack)
8.      for i in range(len(orphan_load_stack)):
9.          load = orphan_load_stack[0]
10.         orphan_load_stack = orphan_load_stack[1:]
11.         successful_assignment, links_to_add, links_to_remove = load.heuristic_domino_flick(currently_active_links)
12.         reset_link_checks()
13.         if successful_assignment:
14.             currently_active_links.difference_update(links_to_remove)
15.             currently_active_links.update(links_to_add)
16.         else:
17.             orphan_load_stack.append(load)
```

Pseudocode Block 2.2: Heuristic Solver overview.

First, in lines 1-5, variables are defined: *currently_active_links*, *orphan_load_stack*, and *previous_orphan_count*. The set *currently_active_links* is just a container to store which links have been used in power assignments so far. The list *orphan_load_stack* is a list with all loads in it, which is sorted by order of priority (from highest level to lowest). Loads that do not yet have assignments are referred to in this pseudocode as "orphans." The orphans that have higher priority are found assignments before other loads in the list. The integer *previous_orphan_count* is a variable that will be used to measure when the algorithm has completed.

Line 6 has the conditions by which the main loop of this algorithm will continue to execute, wherein each pass is another attempt to find a source to match each of the orphaned

loads. A new pass will be cycled through if two conditions are met: first, that there are actually orphaned loads remaining (*orphan_load_stack*), and second, that there has been a successful assignment since the last time we tried to assign all the remaining loads (*len*(*orphan_load_stack*) < *previous_orphan_load_count*).

Before starting the loop to attempt to assign every load in the *orphan_load_stack* (line 8), the current number of orphan loads must be accounted for (line 7), so that we know whether any were assigned when it's finished. The function *range* creates a list of numbers from 0 to n-1, where n is the integer input. In this case, n is the number of items in *orphan_load_stack*. The reason to not simply write "for *load* in *orphan_load_stack*:" is because *orphan_load_stack* is going to be modified inside this loop, so it may acquire and lose elements as it's executing.

The operation taking place in lines 9 and 10 is essentially to pop the first item from *orphan_load_stack* and assign it to *load*. Why don't we just reverse the list so we can pop it from the end with the *pop* operation? Because then we'd have to insert each load that couldn't be assigned into the front of the list, which is slower than *append*. But this otherwise arbitrary choice has to do with the idiosyncrasies of the Python language, and changes nothing about the algorithm generally.

Line 11 calls the operation function, *heuristic_domino_flick*. This function will be covered in more depth later, but all we should know in this block of code is that it is what determines whether *load* can find a source to supply it with power in the network. It takes as an argument *currently_active_links*, because it must know which links are available and which links that could

be newly assigned are potentially incompatible with links that are already assigned. The function returns a Boolean, *successful_assignment*, and two lists, *links_to_add* and *links_to_remove*.

If *successful_assignment* is True, then *load* has been able to find a source in the network to supply it. In that case (lines 14 & 15), *currently_active_links* is updated first to remove links that were active before the assignment but now are not, and then to add links that were not active before the assignment but now are. If *successful_assignment* is False, then the current *load* goes back on the end of the *orphan_load_stack*, and it will have another chance to be matched if there have been any successful matches on this pass through all the orphaned loads (and hence, which links are active and which are inactive gets shuffled around, which could allow for the activation of links that were previously incompatible with the links that were currently active).

There is one step that takes place after this pseudocode block—the loads connected to links that have been marked "active" are themselves marked "active." Before all the assignments are finalized, only the link objects are marked as active or not active, and the load objects are left marked inactive. This is because it's not necessary to mark the loads to find the solution, so we save a small amount of processing power by switching their states at the end.

Here, we should look more closely at line 11, and see the details of the function being called. This pythonic pseudocode block is a function of the load object:

```
1.  def heuristic_domino_flick(currently_active_links):
2.      successful_route = False
3.      links_to_add = set()
4.      links_to_remove = set()
5.      for link in ordered_links:
6.          if not link.checked and not currently_active_links.intersection(link.incompatible_with):
7.              successful_route = link.fall_on_source([], currently_active_links)
8.              if successful_route:
9.                  links_to_add = set(successful_route[0::2])
10.                 links_to_remove = set(successful_route[1::2])
11.                 break
12.     return (successful_route, links_to_add, links_to_remove)
```

Pseudocode Block 2.3: *heuristic_domino_flick* function.

This is the function that sets off a chain reaction, like dominoes. This is the distributed search for the augmented path between source and load. Augmented paths are paths which do not contain cycles and use only edges where the source has enough capacity for the loads they're being asked to provide. Let's break it down line by line.

In lines 2-4, variables are defined that will be used in this algorithm: *successful_route*, *links_to_add*, and *links_to_remove*. The variable *successful_route* will eventually take the value of a list of links in a successful "route," with even-indexed links being links that should be activated and odd-indexed links being links that should be deactivated, but it is initialized to be False in case a route was not found. The sets *links_to_add* and *links_to_remove* are the sets of links that will be added to or removed from *currently_active_links* if the assignment is successful, respectively. They will take their values from *successful_route*.

Line 5 begins the for loop that iterates through every link attached to this load, in a special order. This order is determined as part of the network discovery phase, as it falls into the category of "things that are generally known about the network." This may seem unfair to relegate it to

preprocessing, but the ILP Solver also uses a list called *incompatibilities*, which is a list of all pairs of incompatibilities in the network, and that list is put together during the network discovery phase as well.

The order of links in *ordered_links* is this: for the list of links attached to the load or source in question, sort it first, from least to most, by the number of incompatibilities that that link has with other links in the network. Then, iterate through the list in order, and for each link, if that link is a "choking link," send it to the end of the list. A "choking link" is a link that, for any source in the network, is incompatible with every link attached to that source. This ordering favors choosing links that allow more other links to be potentially activated.

Line 6 stipulates the two conditions for which we can proceed with testing to see if the link in question will be able to be activated: first, to make sure that the link has not already been checked as part of this call to *heuristic_domino_flick*, and second, to make sure that none of the links that are already active are incompatible with it.

Line 7 is the first call to *fall_on_source* in a chain of potentially many loads calling *fall_on_source* in a depth-first-search for an augmented path. Sources have a complimentary function call named *fall_on_load*. We will examine *fall_on_source* and its counterpart shortly, but for now we must simply know that this function returns either a list of links in a successful route or False, and that return value is assigned to *successful_route*.

Lines 8-11 comprise a conditional statement that if the link was able to be assigned, *links_to_add* is assigned to be a set of every even-indexed link in *successful_route*, and *links_to_remove* is assigned to be a set of every odd-indexed link in *successful_route*. Once that

information is gleaned, the *break* statement is executed on line 11 to break out of the loop, because no further links need to be checked for availability to find that augmented path. Then *links_to_add* and *links_to_remove* are returned (line 12) and the function is concluded.

In this explanation, we have been opening a series of nested black boxes for functions that contain part of the process to find a load's augmented path, and there is still further to go. Let's look more closely at the *fall_on_source* function of the link object:

```
1.  def fall_on_source(ghost_route, currently_active_links):
2.      checked = True
3.      new_ghost_route = ghost_route + [self]
4.      successful_route = source.domino_drop(new_ghost_route, currently_active_links)
5.      if successful_route:
6.          active = True
7.          source.supply_lines.update([self])
8.          source.ordered_supply_lines = [link for link in source.ordered_links if link in source.supply_lines]
9.          load.supplied_by = self
10.     return successful_route
```

Pseudocode Block 2.4: *fall_on_source* function.

The two arguments passed into this function (line 1) are *ghost_route*, which is a list of links in the path that will be built one step at a time, and *currently_active_links*, which has already been discussed. The initial value for *ghost_route* that is passed in by *heuristic_domino_flick* is an empty list. The first addition is in line 3, where *self* is appended to it. Because this is a function of the link object, the *self* being referred to here is the very link that is executing this pseudocode. Line 2 simply marks this link as checked, so it is not examined again during this path search.

Line 4 calls *domino_drop*, a function of the source object. This is where the black box is in this function, because all we need to know at this point is that we pass in *new_ghost_route* and

currently_active_links, and it returns a route of links that can be activated or deactivated if the assignment is successful, and False if not. In either case, this value is assigned to *successful_route*.

If there hasn't been a successful assignment, then the conditional statement starting on line 5 is skipped over and the value False is returned in line 10, indicating an unsuccessful assignment. If there is a route returned, though, then the conditional statement is executed.

Since the function *fall_on_source* is only ever called by a load, then if the assignment is successful, that means the load is able to be supplied by the source. That's why this link is switched to active (line 6) and the source that is connected to it adds this link to its set of links that it is supplying power through (line 7). Line 8 just updates source's list of links it's supplying according to the order of *ordered_links*, which is a step whose importance will become more apparent in the next code block explanation. Line 9 updates the load's link that's supplying it to this link.

Now, let's go one level deeper into the black box, and look closer at what's going on with the function *domino_drop* on line 4:

```
1.  def domino_drop(ghost_route, currently_active_links):
2.      if len(supply_lines) + 1 <= capacity:
3.          return ghost_route
4.      successful_route = False
5.      for link in ordered_supply_lines:
6.          if not link.checked:
7.              successful_route = link.fall_on_load(ghost_route, currently_active_links)
8.              if successful_route:
9.                  break
10.     return successful_route
```

Pseudocode Block 2.5: *domino_drop* source function.

Here is, potentially, the end of the black boxes. If this source has capacity enough capacity to support another load (comparison on line 2), then the route is successful. The *ghost_route* is returned to *successful_route* in the layer above it, and so the whole algorithm is concluded. (It should be noted that if the loads did not simply have unit capacity, then this is where the specific load requirement would be tested and accounted for.)

However, if this source does not have sufficient capacity to add one more load to its supply, then that is not the end of it. It is possible that, if this source simply asked for one of its loads to find another supplier, then it would be able to accommodate this load. And so in lines 4-10, this test is performed.

Line 4 initializes *successful_route* to False. It will be given the value of a list of the links in a path if there is a successful assignment. Otherwise, it is returned as False in line 10.

Line 5 begins the for loop for each link connected to this source along which it is supplying power. If the link has not already been checked as part of this algorithm (line 6), then we go one step further into a black box function looking for an answer (line 7). This function, *fall_on_load*, takes the *ghost_route* as has been so far proposed, and *currently_active_links* so that information about incompatible links can be considered. Either a successful route or False is returned to the variable *successful_route*. We don't need to consider whether any of these links we're testing are incompatible with the links in *currently_active_links*, because they should already be members of that set! If a successful route is found (line 8), then no further links need to be tested (line 9).

Here we take a closer look at *fall_on_load* on line 7, going again another step into these black box functions looking for an answer:

```
1. def fall_on_load(ghost_route, currently_active_links):
2.     checked = True
3.     new_ghost_route = ghost_route + [self]
4.     successful_route = load.domino_drop(new_ghost_route, currently_active_links)
5.     if successful_route:
6.         active = False
7.         source.supply_lines.difference_update([self])
8.         source.ordered_supply_lines.remove(self)
9.     return successful_route
```

Pseudocode Block 2.6: *fall_on_load* function.

This function is a parallel to *fall_on_source*. It is another function of the link object. Instead of testing to see if this link can be activated, however, this function tests to see if the link can be deactivated. If it can be deactivated, that means the load connected to it (that was previously supplied by the source that called this function) has found a new source to supply it.

First the load is marked checked (line 2) so that it is not considered again during this pass. The function *domino_drop* is then called, this time in the load object (line 4). This function is different from the *domino_drop* that is in the source object, and constitutes yet another black box that will be examined in closer detail further on. For now, all that is important is that it takes as parameters the ghost route (with this link added to it, line 3) and *currently_active_links* for incompatibility information, and it returns either the list of links in a successful path if the assignment is successful or False, which is a value assigned to *successful_route*.

If no route has been found from *domino_drop*, then the False value of *successful_route* is returned directly (line 9). However, if there has been a successful augmented path assignment (line 5), then this link is deactivated (line 6). The connected source also removes it from its set of links through which it is supplying power (line 7) and from its ordered list (line 8) before the function concludes and returns the route.

Here we will peer even one step deeper into the black box function on line 4. This *domino_drop* is a function of the load object:

```
1.  def domino_drop(ghost_route, currently_active_links):
2.      ghost_links_to_add = set(ghost_route[0::2])
3.      ghost_links_to_remove = set(ghost_route[1::2])
4.      projected_linkset = currently_active_links.copy()
5.      projected_linkset.difference_update(ghost_links_to_remove)
6.      projected_linkset.update(ghost_links_to_add)
7.      successful_route = False
8.      for link in self.ordered_links:
9.          if not link.checked and not projected_linkset.intersection(link.incompatible_with):
10.             successful_route = link.fall_on_source(ghost_route, currently_active_links)
11.             if successful_route:
12.                 break
13.     return successful_route
```

Pseudocode Block 2.7: *domino_drop* load function.

This function must perform similarly to *heuristic_domino_flick*, in that it must test every link that is attached to this load to see if it can be supplied power from that link's source. But not every link is compatible with the links that will be active if the proposed route is chosen. Because at this point in the Heuristic Solver the proposed route includes switching some links off that are currently on and switching some links on that are currently off, it is no longer simply incompatibility with the links in *currently_active_links* that must be accounted for.

That is why a new set is created in lines 2-6 called *projected_linkset*, because these are the links that are going to be active if this route is successful. It is first created as just a copy of *currently_active_links* (line 4). Then, all the links that will be deactivated in this route are removed from it (lines 3 and 5), and all the links that will be activated in this route are added to it (lines 2 and 6).

As in these other functions we've looked at, *successful_route* is the variable that will measure whether a successful route has been found. It is initialized as False (line 7), but will be updated to a list of links in the successful route if one can be found before it is returned at the conclusion of the function on line 13.

Line 8 begins the loop through all links (in the precomputed proper order), checking for a link to reassign this load to. Only the links that would be compatible with the other active links in this route should be considered, and they should not have already been considered previously in this algorithm. This is the test on line 9. Line 10 contains another black box function, but it's one that we've seen before. It is another call to *fall_on_source*, which takes *ghost_route*, *currently_active_links*, and returns either a list of links or False to *successful_route*. Of course, if a successful route is found, then no more links need to be tested (lines 11 and 12).

At this point, the algorithm has come full circle. The logic can be followed through *fall_on_source* to *domino_drop* to *fall_on_load* to *domino_drop* to *fall_on_source* again, and on and on. Ultimately, the search must conclude in one of two ways—either a source with free capacity will be able to supply another load, or all the potentially compatible links that can be checked will be checked without finding a successful route. So if a successful route is to be found, it will be because the condition in line 2 in the source object's *domino_drop* function evaluates to True.

It should be noted that before the possibility of multiple links between each source and load was accounted for, there were no link objects at all and the sources and loads each called a different version of each other's *domino_drop*. A list of incompatibilities was passed in, because

they were kept track of at the level of the network object, and links were simply defined by the source/load pair. Having a link object allows for multiple paths between the same source and load, and makes it easier to keep track of incompatibilities. Why it's important to account for the possibility of multiple links between the same source and load will become apparent when we discuss the synthetic networks created to test this algorithm.

It should be noted too that in this algorithm, a load that has already been assigned power supply by a source will never become unsupplied. Which source supplies it may change, as a different link may need to be used in order to be compatible with other links in the network, or it may have lines to other sources with more capacity that other loads do not have access to.

There are many steps to this algorithm, but perhaps it will seem clear if we walk through it using our example network. First, all the links are checked to make sure they are inactive, and then the *orphan_load_stack* is populated in order of lowest priority load to highest: ('L5', 'L4', 'L3', 'L1', 'L2'). Then we pop them off one by one from the front and try to assign a load to them. There are 5 orphans in this pass.

L5 is first. It looks at the first link in its *ordered_links* list, 1->5_1. There are no other links that are currently active, so there are no incompatibilities. S1 has 2 capacity available, which is plenty to accommodate L5. The link is activated.

Next is L4. It checks the first link in its *ordered_links*, 1->4_1. This link is not incompatible with the currently active link, 1->5_1. S1 has 1 capacity available, so it is able to supply L4. The link is activated.

L3 is third in line. The link 1->3_1 is not incompatible with any links in *currently_active_links*, so we test the capacity of the connected source. S1 has 0 capacity by this point, so if S1 is going to supply L3, it needs to reassign one of the loads it's feeding to get power from another source. It checks the load at the other end of 1->4_1, L4. This load must look at links connected to it that haven't already been checked during this pathfinding process for another supply line. It checks the link 2->4_1, which must not be incompatible with 1->3_1 or with 1->5_1. It isn't, so we check the source on the other end of it for availability. S2 has capacity of 2, so it can indeed accommodate L4. The path is therefore successful, so 2->4_1 is activated, 1->4_1 is deactivated, and 1->3_1 is activated.

L1 is the next load to check. It looks at 2->1_1, which is not incompatible with any of the loads in *currently_active_loads*: ['1->5_1', '2->4_1', '1->3_1']. The source on the other end, S2, has capacity of 1 remaining. So it is able to supply L1 through 2->1_1, and the link is activated.

Last in line is L2. It can't choose to check 2->2_1 because that link is incompatible with the link 1->3_1, which is currently active. So it checks 1->2_1 instead. The source at the other end of it, S1, is already at maximum capacity, but it might be able to accommodate L2 if it reassigns one of them. First it tries to find a new supplier for the load at the other end of 1->5_1. The link 1->5_2 is compatible with 1->2_1, 2->4_1, 1->3_1, and 2->1_1, and it has not been checked yet, so we look at the source on the other end of it. S1 is already at maximum capacity, so we have to try to reassign a load that it is feeding. 1->5_1 has already been checked, so instead we examine 1->3_1. The only other link attached to L3 is 2->3_1, which is incompatible with 2->4_1. So L3 can't find an alternate. S1 isn't supplying any other loads, so it can't try to find alternates for any of them to accommodate L5. L5 doesn't have any other links attached to it that haven't been

checked, so it cannot be reassigned. S1 isn't supplying any other loads, so it can't try to find alternates to accommodate L2. L2 doesn't have any other links it could try, so the result is that it is unable to find an assignment. It goes to the end of the *orphan_load_stack* list.

Now there is 1 load in *orphan_load_stack*, which is less than the 5 loads before we iterated through all of them looking for source assignments. This means we've made progress. So we're going to iterate through *orphan_load_stack* again.

The single item in *orphan_load_stack* is L2. Because nothing has changed since the last time it tried, the process that L2 goes through looking for a source to supply it is identical to the process it went through in the last iteration through all the orphans. So again, it fails to find a path. It is moved to the end of *orphan_load_stack*.

Now, there is still 1 load in *orphan_load_stack*. This is the same number of orphaned loads that there were during the last iteration. This means no progress has been made, and it is futile to continue attempting to make further assignments. Here is the final solution given to us by Heuristic Solver for this network:

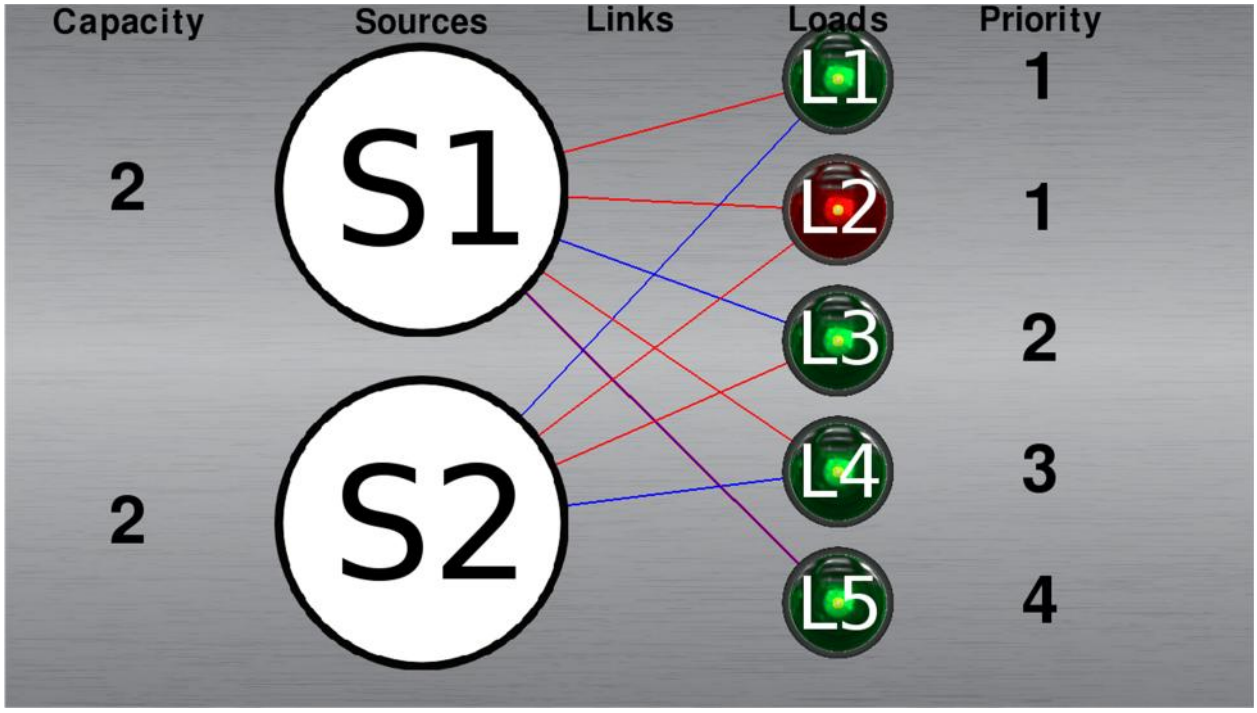

Figure 2.4: Example network Heuristic Solution.

In tabulated form, here are the active links:

Links	Active
1->1_1	False
1->2_1	False
1->3_1	**True**
1->4_1	False
1->5_1	**True**
1->5_2	False
2->1_1	**True**
2->2_1	False
2->3_1	False
2->4_1	**True**

Table 2.4: Example network active links, Heuristic Solution.

The first noticeable thing about this solution when compared to the ILP Solver's solution for the same network is that the number of loads that have power are the same:

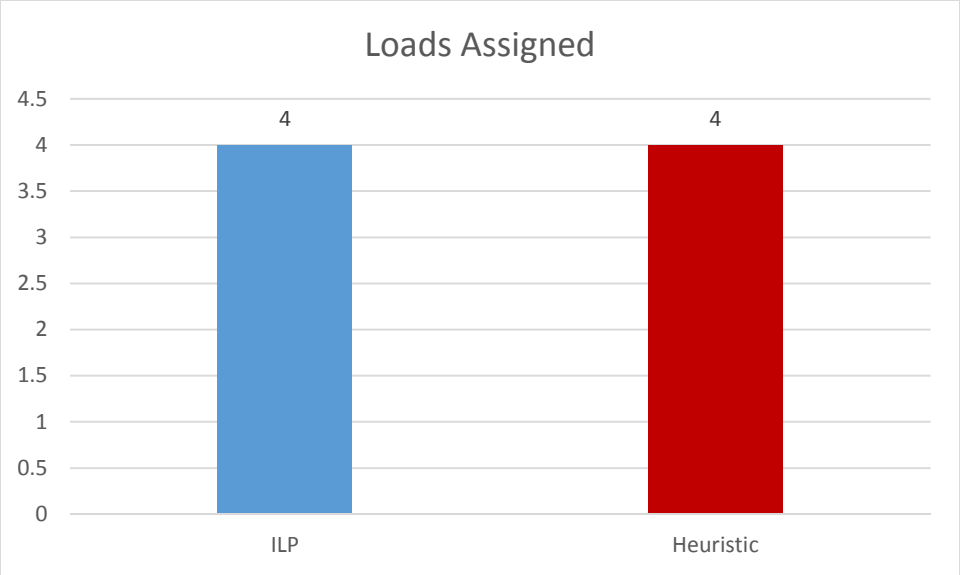

Figure 2.5: Example network loads assigned.

The loads that were assigned are not identical—the ILP Solution assigned power to L5, L4, L3, and L2, while the Heuristic Solution assigned power to L5, L4, L3, and L1. L1 and L2 are each priority level 1, which is the lowest level priority in the network. It should be unsurprising that the Heuristic Solution's composite score is the same as the ILP Solution:

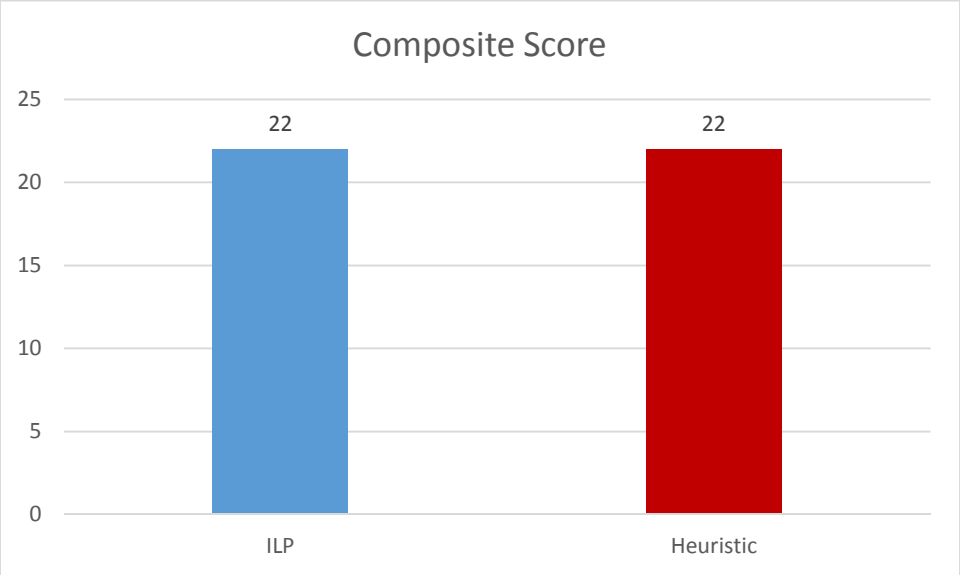

Figure 2.6: Example network composite score.

There is one major, important difference between these two solutions, and that is the time it took for each to be computed:

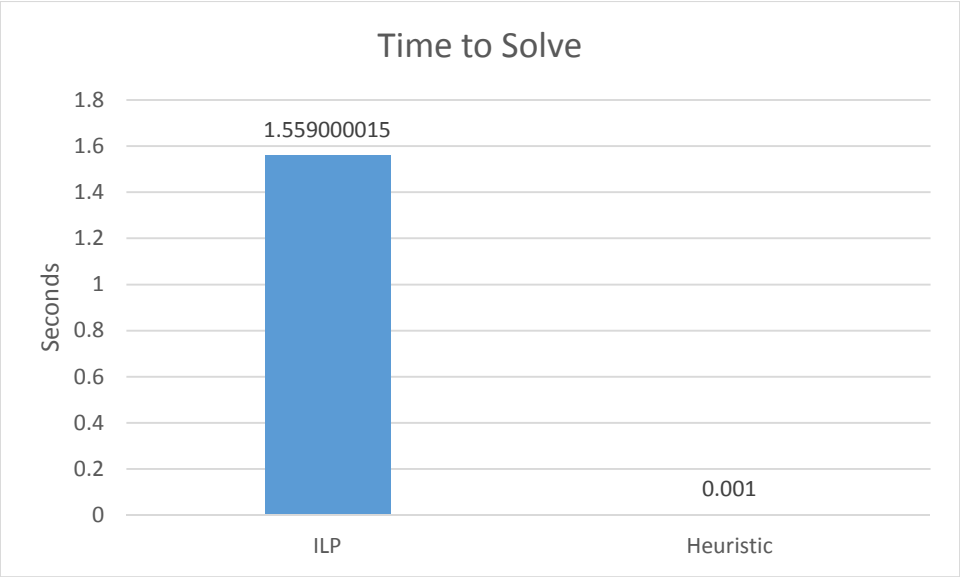

Figure 2.7: Example network time to solve.

The ILP Solver took 1,559 times longer than the Heuristic Solver to find a solution that is of equal quality. Here we begin to see the usefulness of the Heuristic Solver.

3. FLATTENING THE DISTRIBUTION NETWORK

So far, we have looked at one example of a power distribution network in a flattened view. A flattened view of a power distribution network is an abstraction that breaks it down to its functional components: sources, loads, and the pathways that connect them. In this section, we will see how this is achieved by translating a practical example network that has been designed for educational use and testing [11], which we will call the Benchmark Network, to this flattened view. This network is structured radially, which is typical for a power grid application. Power distribution networks in electrical grids are usually designed to be radial for safety and simplicity, but mesh configurations are more reliable [12]. That is why we will also generate artificial grid networks, which we will call Synthetic Networks, and translate them into the flattened view as well. The mesh style might be more typical in a network where reliability is important, which may be likely to be a context in which the Heuristic Solver presented in this thesis is implemented.

First, here is the Benchmark Network:

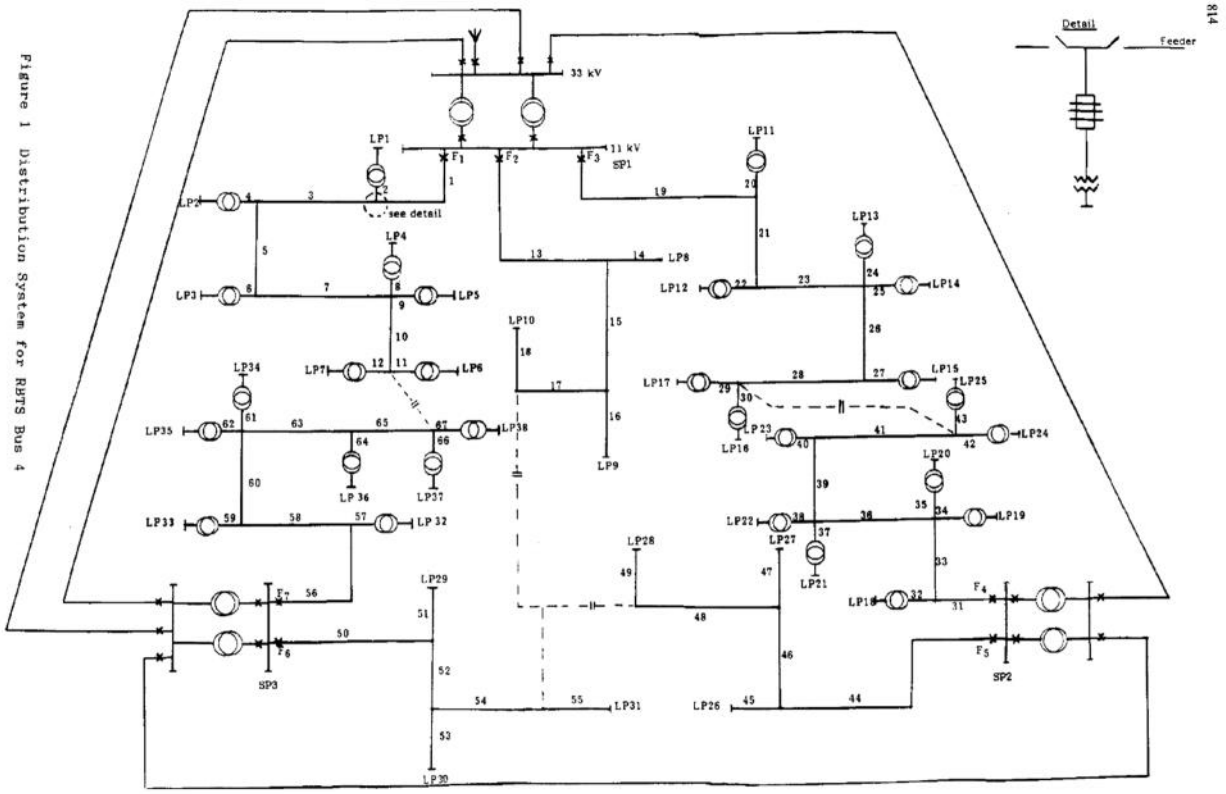

Figure 3.1: Benchmark Network from [11].

There are many components of this network that can be abstracted, namely, anything that is not a power supply, a transmission line, a switch, or a load. Here are all the generators and loads:

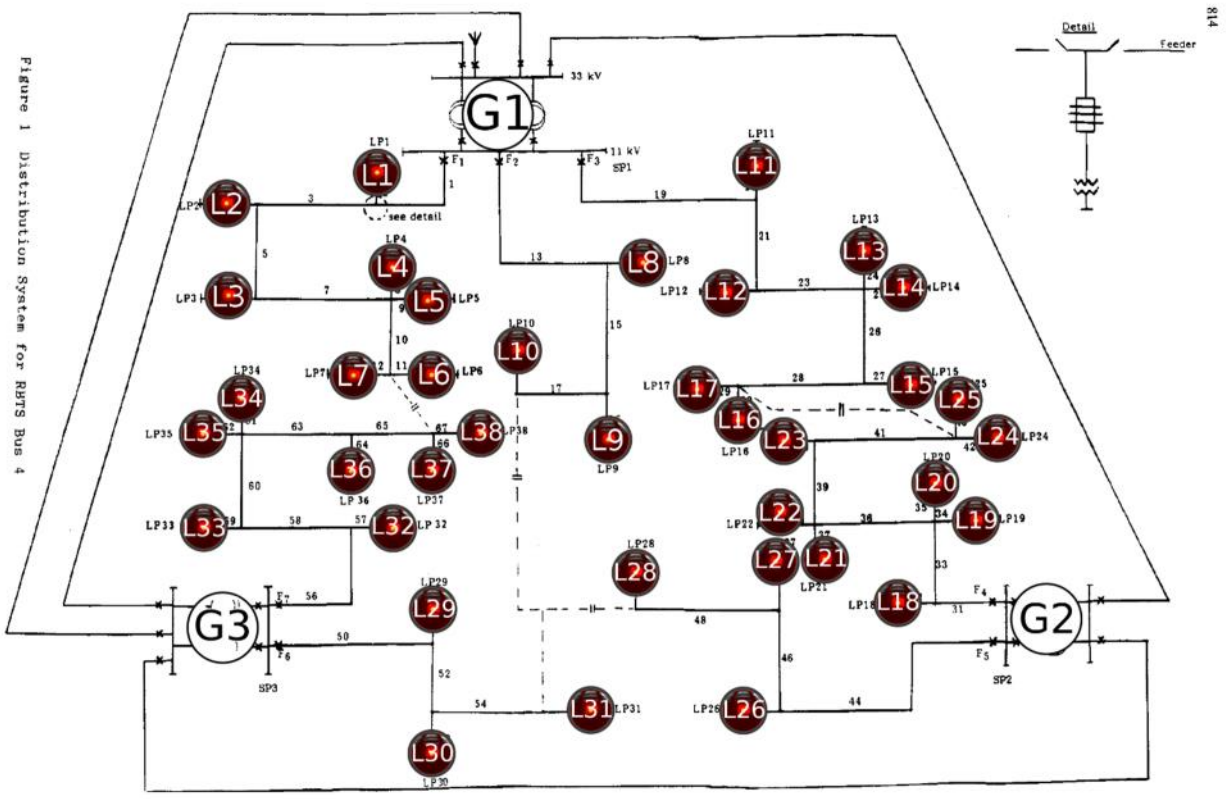

Figure 3.2: Benchmark Network with generator and load labels.

The lines should be simplified as well. In order to not have a trivial solution to this circuit, we are going to assume that each load can only be supplied by one source. Whatever transformers there are in the circuit potentially allowing multiple sources to feed into them, they are not going to be considered in this test:

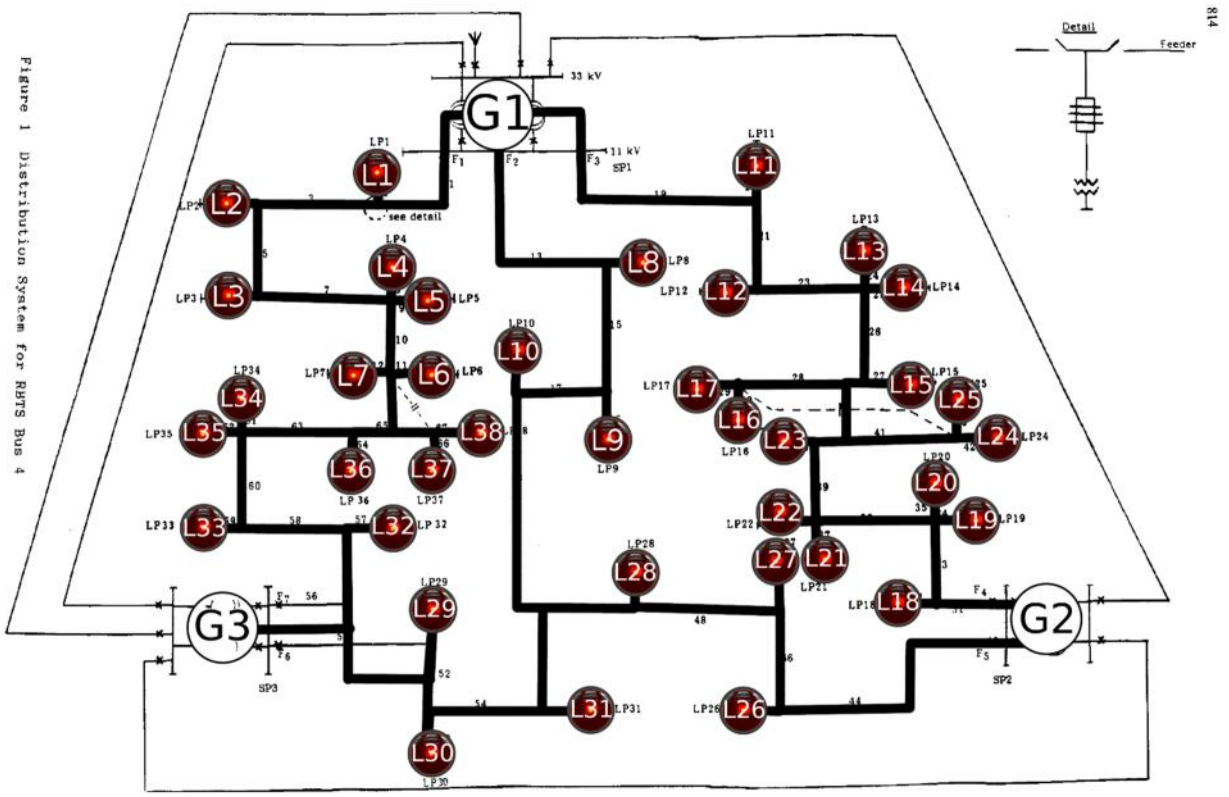

Figure 3.3: Benchmark Network with simplified lines.

Note that four our purposes, the lines between generators don't affect the topology. We should, however, include switches where they are shown on the diagram:

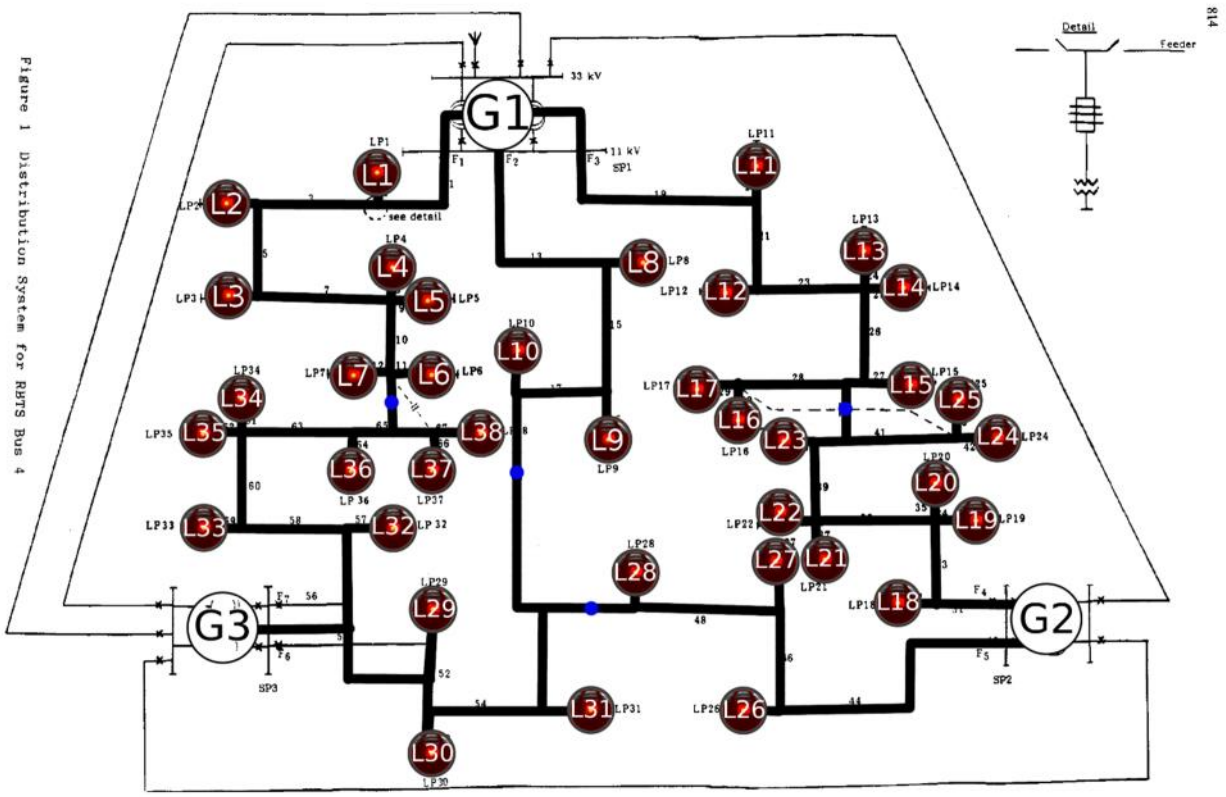

Figure 3.4: Benchmark Network with branch connector switches.

This diagram is not necessarily fully detailed, because there are likely to be switches at each source output and load input. This is important for our flattening process, so we include them now:

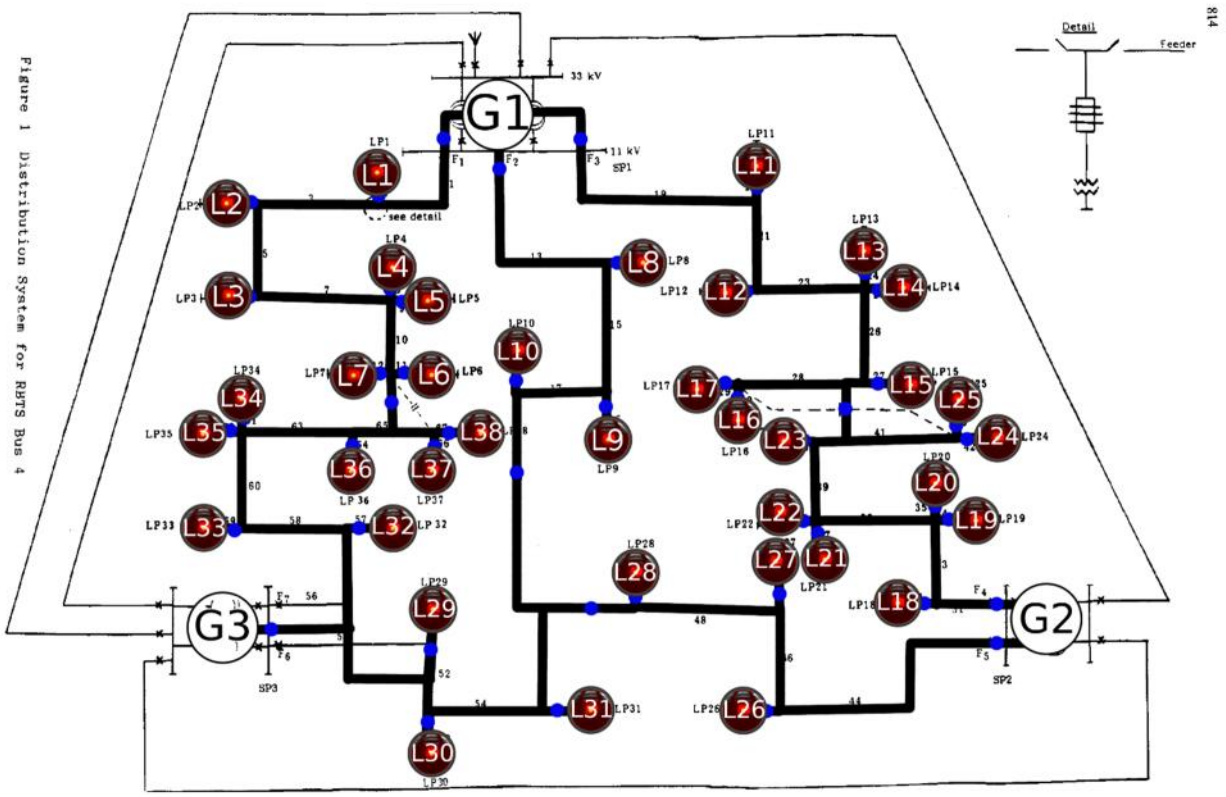

Figure 3.5: Benchmark Network with all switches.

We have abstracted this network sufficiently to remove the original circuit diagram and look only at the conceptualization we've made:

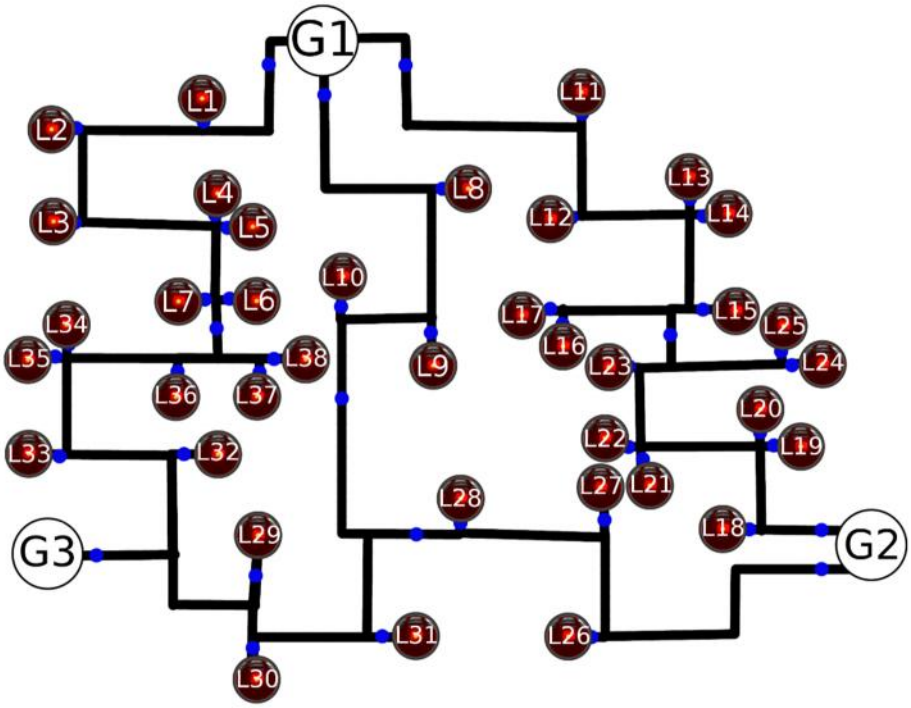

Figure 3.6: Benchmark Network, simplification only.

At this point, we should give names to the switches as well. The switches at each load are not labeled here for the sake of keeping the diagram clean, but they follow the naming scheme of *nL1*, *nL2*, *nL3*. The other switches are named as follows:

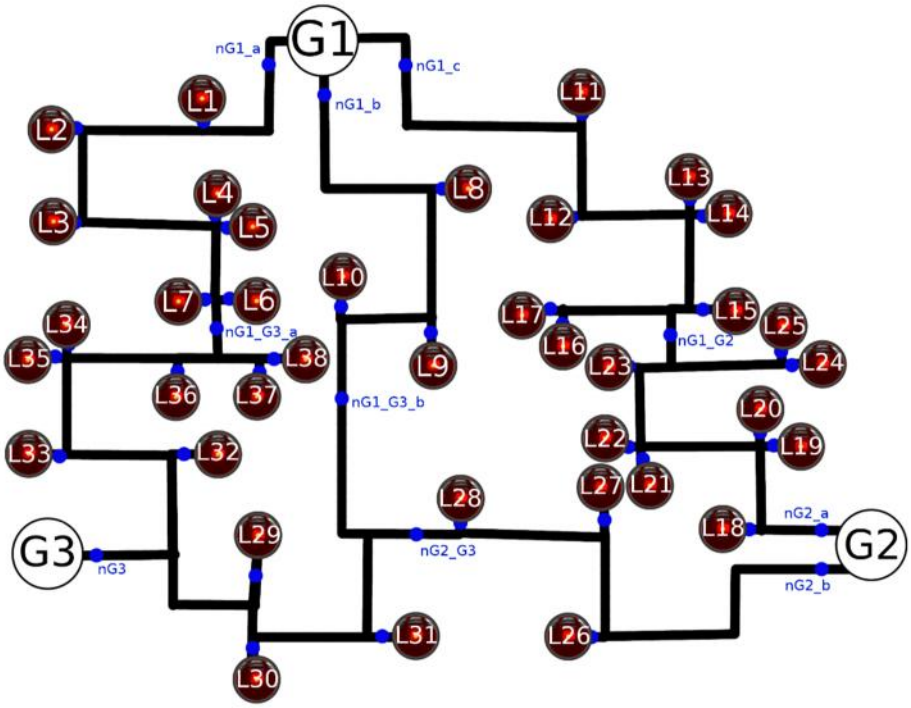

Figure 3.7: Benchmark Network with switch labels.

Further abstraction must be done here. If we were to look at which switch is linked to which other switch, we would see, for example, that *nG1_a* is connected to *nG1_G3_a*, and *nG1_G3_a* is connected to *nL2*. Is *nG1_a* also connected to *nL2*? That cannot be the case, because then if there is a line activated from G3 through *nG3, nG1_G3_a, nL1*, to L1, then there shouldn't be any conflict with G1 supplying L2 through *nG1_a* and *nL1*. So in order to prevent these paths from crossing, we must say that logically, *nG1_a* is connected only to *nG1_G3_a*, and that *nG1_G3_a* is connected to *nL1*, *nL2*, *nL3*, *nL4*, *nL5*, *nL6*, and *nL7*.

However, if we follow that logic on the other side of the switch, then *nG3* is connected to *nG1_G3_a* as well, and *nG1_G3_a* is connected too to *nL29*, *nL30*, *nL31*, *nL32*, *nL33*, *nL34*, *nL35*, *nL36*, *nL37*, and *nL38*. In this case, if G1 supplies L1 through *nG1_a*, *nG1_G3_a*, and *nL1*, then G3 cannot choose *nG1_G3_a* in its path to any other nodes. It could of course choose *nG1_G3_b* or *nG2_G3*, but there is same conflict on the other sides of those switches as there is with *nG1_G3_a*. Therefore, we introduce another abstraction to remedy this:

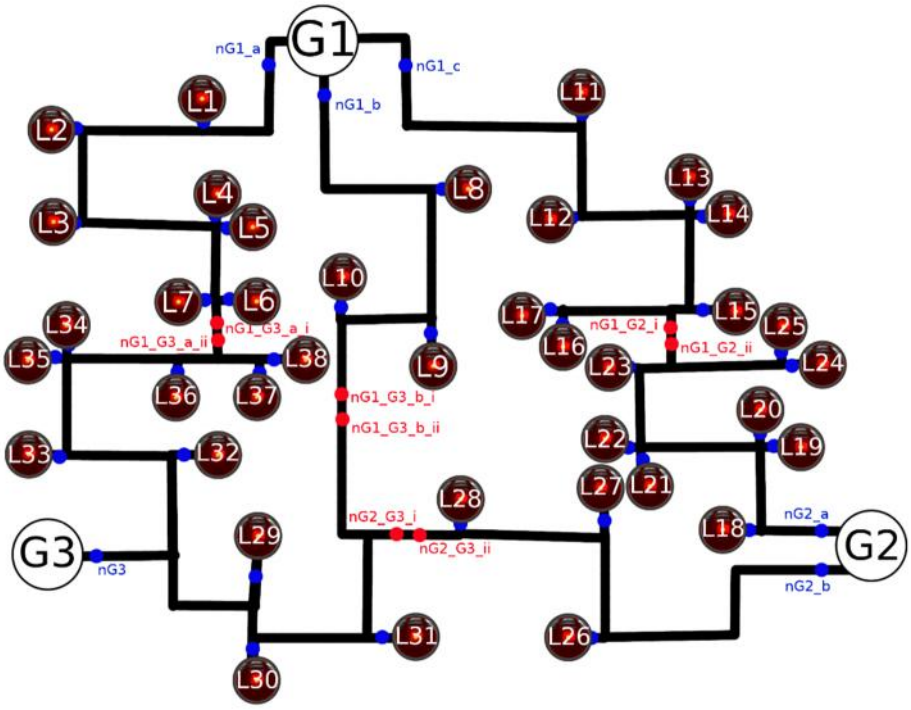

Figure 3.8: Benchmark Network with connector switch abstractions.

Each pair of red connector switches are connected to each other, and each member of the pair is connected to the load switches and generator switch on its side of the divide. We're

still not done yet, however, because logically, if G1 supplies L29 through *nG1_a*, *nG1_G3_a_i*, *nG1_G3_a_ii*, and *nL31*, then G3 can still supply its loads like L30 through a path like *nG3*, *nG2_G3_i*, and *nL30*. This should not be permitted. Therefore, we must add one further layer of abstraction, which is the realization that *nG1_G3_a_ii*, *nG1_G3_b_ii*, and *nG2_G3_i* are all logically the same:

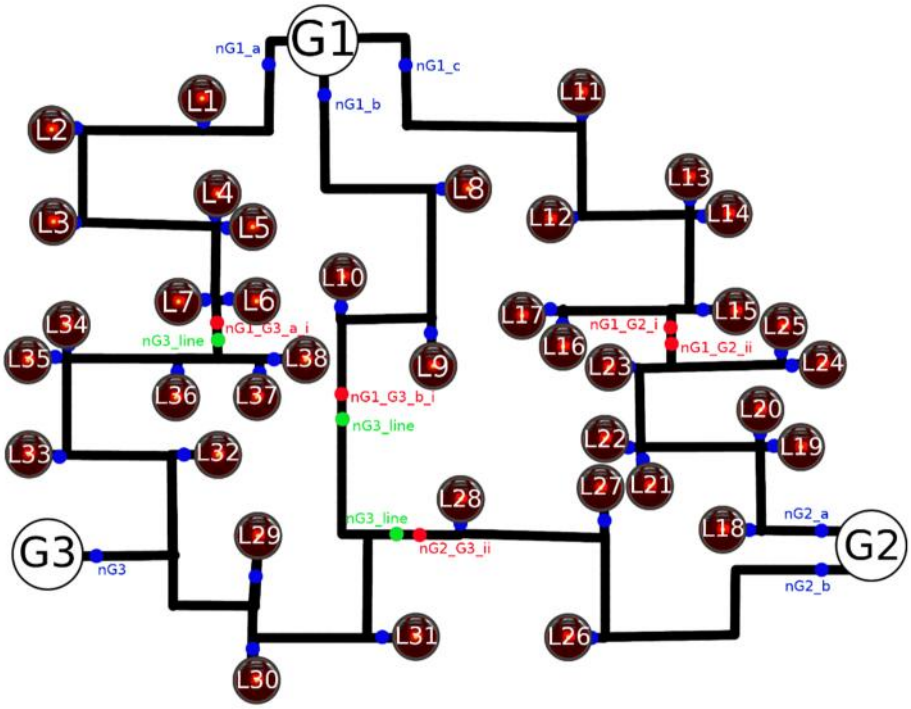

Figure 3.9: Benchmark Network, nG3_line abstraction.

At last, our abstractions are complete, and we can actually redraw this diagram as its logical equivalent in a treelike structure:

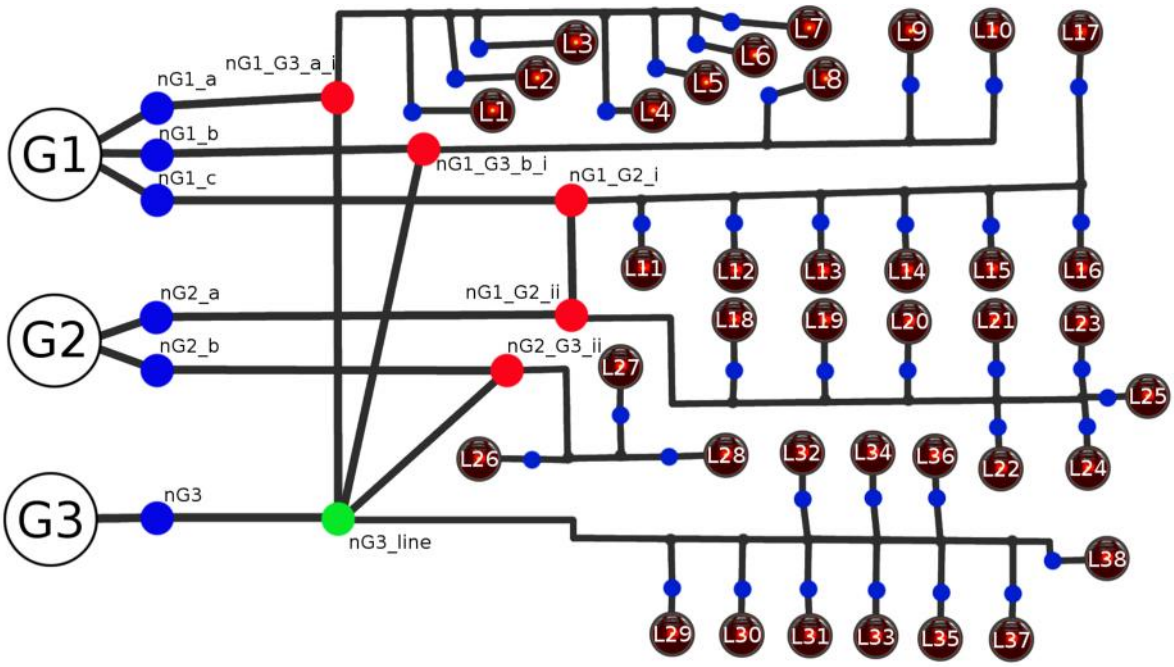

Figure 3.10: Benchmark Network, logical layout.

Now that we have conceptualized this network as a simple topology of sources, switches, loads, and links, we are about halfway finished with flattening the network. First, though, we should make some decisions about units. Here is the source information about this network, taken from [11]:

feeder number	load points	feeder load, MW average	feeder load, MW peak	number of customers
F1	1-7	3.51	5.704	1100
F2	8-10	3.5	5.705	3
F3	11-17	3.465	5.631	1080
SP1 Totals		10.475	17.040	2183
F4	18-25	4.01	6.518	1300
F5	26-28	3.0	4.890	3
SP2 Totals		7.01	11.408	1303
F6	29-31	3.5	5.705	3
F7	32-38	3.595	5.847	1290
SP3 Totals		7.095	11.552	1293
	TOTALS	24.58	40.00	4779

Figure 3.11: Benchmark Network source information from [11].

The individual feeder outputs are listed separately as F1, F2, F3, etc., but we decide to treat each source as one unit, so we should look at the total capacity supply for each. Although we could do these calculations with “peak” information, which would be normal for a situation where we are concerned about making things failsafe, let’s assume that the solution we’re finding is under normal operating conditions and pick the “average” values instead for each source.

Now, if we were going to implement our solvers with precision down to the watt, we could easily assign capacities to be as they are listed in this figure. Instead, we are going to round to the nearest megawatt. The capacities of our sources are thus as follows:

Source	Capacity
G1	10
G2	7

G3	7

Table 3.1: Benchmark Network source capacities.

Here is some information about loads in the Benchmark Network, also from [11]:

number of load points	load points	customer type	load level per load point, MW average	load level per load point, MW peak	number of customers
15	1-4, 11-13, 18-21, 32-35	residential	0.545	0.8869	220
7	5, 14, 15, 22, 23, 36, 37	residential	0.500	0.8137	200
7	8, 10, 26-30	small user	1.00	1.63	1
2	9, 31	small user	1.50	2.445	1
7	6, 7, 16, 17, 24, 25, 38	commercial	0.415	0.6714	10
TOTALS			24.58	40.00	4779

Figure 3.12: Benchmark Network load information from [11].

There are five categories of loads, each belonging to one of three customer types. Again, if we were going to deal with more precise units, we would be able to deal with 0.545 or 1.500 MW, but we're keeping things simple and abstract, so we're going to approximate these values as simply 1 MW each.

There is also the issue of load priorities, which we have some liberty to decide because it is not specified anywhere in the reference document. The priority assignments we make can be justified by the descriptions of customer type and number of customers, but they are ultimately arbitrary and for the purpose of demonstration. As such, our priority assignments are these:

Group	Loads	Priority
A	L1, L2, L3, L4, L11, L12, L13, L18, L19, L20, L21, L32, L33, L34, L35	3
B	L5, L14, L15, L22, L23, L36, L37	2
C	L8, L10, L26, L27, L28, L29, L30	1

D	L9, L31	1
E	L6, L7, L16, L17, L24, L25, L38	4

Table 3.2: Benchmark Network load priorities.

With these parameters decided, we can proceed with flattening the topology and then find a power assignment solution with each of the solvers. What remains to be done is iterating through every possible path from each source to the loads it can provide power to, and keeping track of their incompatibilities with other links. To go through this process link by link would be entirely too exhaustive, so we will just examine the different levels of this algorithm, starting with the topmost level:

```
1. for source in sources:
2.     source.find_paths()
3. for source in sources:
4.     for s in sources.difference([source]):
5.         source.catalogue_incompatibilities(s.links)
```

Pseudocode Block 3.1: Flatten network overview.

This algorithm really has two parts—lines 1 and 2, which enumerate the links coming out of each of the sources, and lines 3-5, which name the incompatibilities of those links. First, let's look at the function *find_paths* on line 2, which finds the links coming out of each source:

```
1.  def find_paths():
2.      for entry_switch in self.entry_switches:
3.          entry_switch.tunnel(self)
4.      for load, switchsets in links_by_load.items():
5.          i = 1
6.          for switchset in switchsets:
7.              link_name = str(identifier) + '->' + str(load.identifier) + '_' + str(i)
8.              links[link_name] = switchset
9.              link_incompatibilities[link_name] = set()
10.             i += 1
```

Pseudocode Block 3.2: *find_paths* function.

This is a function of the source object. First, in lines 2 and 3, it finds all the paths for each switch it is connected to. (We assume that there is at least one switch between each a source and any loads, and that there is no link directly between a source and load. Otherwise we would not have control over that part of the supply, and we could remove that load from our topological diagram and subtract the capacity it requires from the source it is connected to.)

Once this has finished, the source should have a dictionary called *links_by_load*, which has a key for every load that this source can reach, and those keys have values called *switchsets*. A *switchsets* variable is a set of *switchset* variables. A *switchset* variable is a set of every switch in a unique path from this source to the load in question. Note that if there are two paths between a source and load that pass through the same set of switches, then they are logically equivalent.

Lines 4-10 serve to name each link according to the naming scheme "S->L_N" as described in the previous section, Power Reconfiguration Algorithms, as well as to catalogue each link in a dictionary with its *switchset*, and to catalogue each link in a dictionary with empty sets that will be filled later with their incompatible links.

Here we should look closer at the *tunnel* function being called on line 3:

```
1. def tunnel(source, pathway=set()):
2.     for switch in connected_switches.difference(pathway):
3.         switch.tunnel(source, pathway.union([self]))
4.     if connected_load:
5.         source.links_by_load[self.connected_load].append(frozenset(pathway.union([self])))
```

Pseudocode Block 3.3: *tunnel* function.

This is a function of the switch object. It is a recursive function—it will call itself on each switch that is connected to the present switch. When it is first called by the source connected to it, the parameter *pathway* defaults to an empty set. Then, for each switch in *connected_switches* that do not include any of the switches already in the *pathway* (to prevent an infinite loop, line 2), this switch calls *tunnel* with itself added to the *pathway* set (line 3).

All along, the parent *source* that is calling this function is passed in to each call of *tunnel*. If a switch is reached that is connected to a load (line 4), then the current set of switches in *pathway* becomes a *switchset* that is added to that key load's *switchsets* set, because it is one unique pathway between that source and this load.

To find the incompatibilities between each of the links is very simple. This is the function *catalogue_incompatibilities* that is called on line 5 of the topmost level of this algorithm:

```
1. def catalogue_incompatibilities(foreign_links):
2.     for link_name_A, switchset_A in links.items():
3.         for link_name_B, switchset_B in foreign_links.items():
4.             if link_name_A != link_name_B and switchset_A.intersection(switchset_B):
5.                 link_incompatibilities[link_name_A].update([link_name_B])
```

Pseudocode Block 3.4: *catalogue_incompatibilities* function.

This function is very simple. It belongs to the source object. Each of the links in the dictionary *foreign_links* passed in (which in this case is the sets of links from each other source) is checked to see if any of the switches that are in its path are also in the path of any links coming out of this source. If so, then that incompatibility is recorded in *link_incompatibilities*, because logically, every switch can only be used by one source. Links attached to the same source are not

checked for incompatibilities with each other because, electrologically, there is no problem with two links passing through the same switches that come from the same source.

Finally, here is what this network looks like flattened:

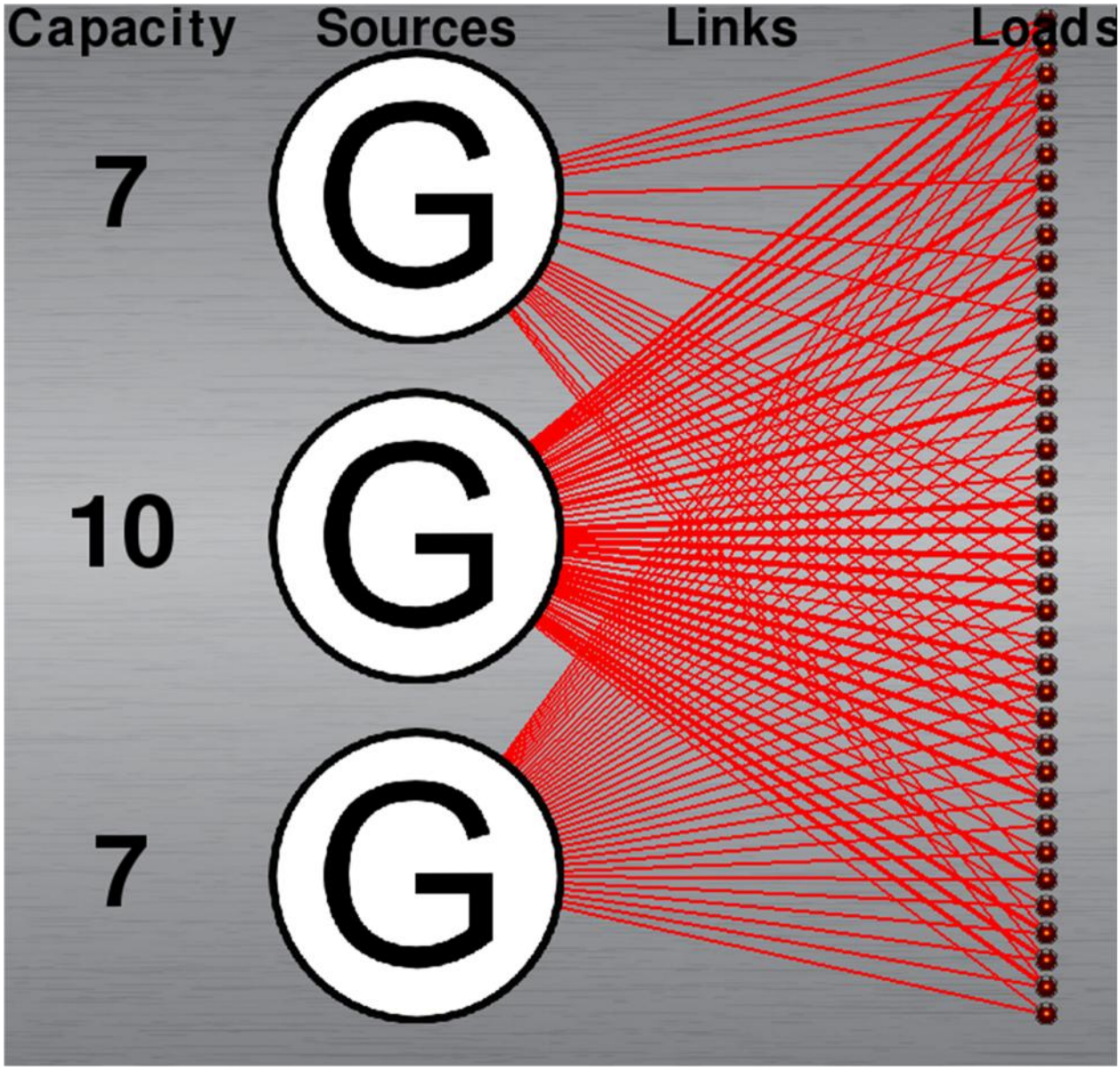

Figure 3.13: Flattened Benchmark Network.

This image that was generated is fairly clustered, and that is due to the sheer number of links and loads in the circuit, but it will become useful in the next section, Experimental Results, where we look at solutions for this network.

It is also necessary to be able to perform this kind of flattening and solving on a network of arbitrary size and structure. That is why we have a process for generating Synthetic Networks, which take on a gridlike structure:

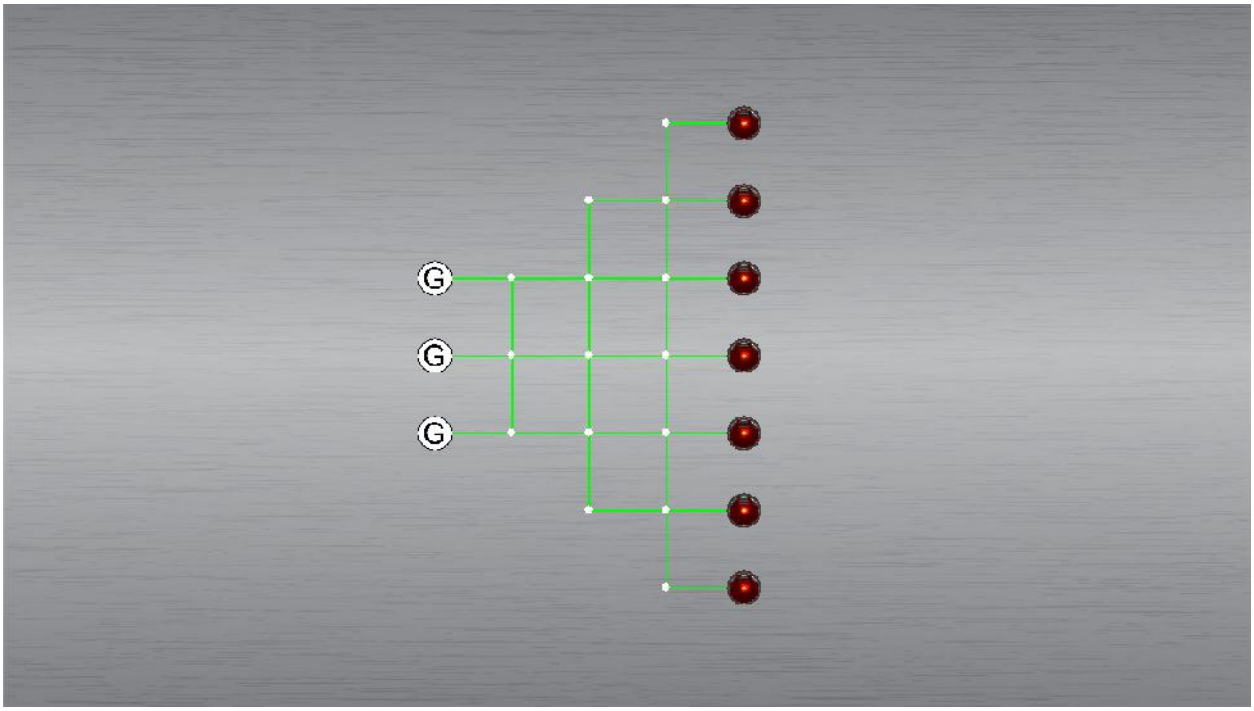

Figure 3.14: Synthetic Network.

The process of translating this Synthetic Network into a flattened view now is very simple—it's the same process as for the Benchmark Network! Each source calls *tunnel* on the switch that is connected to it, and they recursively call *tunnel* until they find all possible routes to loads. Then, incompatibilities are logged based on which links traverse the same switches as which other links. The flattened view for this example looks like this:

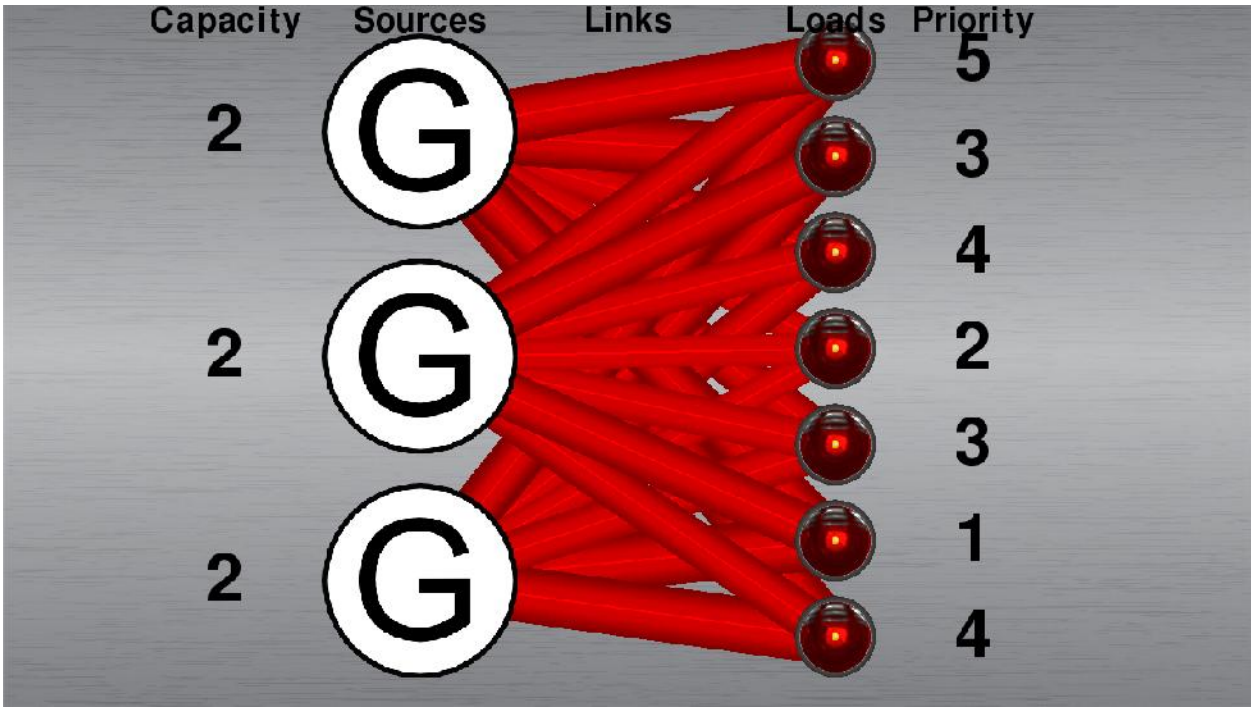

Figure 3.15: Flattened Synthetic Network.

The lines here look incredibly thick because each one is actually dozens of links on top of each other. That's because there are so many possible routes between each source and load, but again, this view will have some use in the next section where solutions for this network are shown.

4. EXPERIMENTAL RESULTS

Now that we have seen how the ILP Solver and the Heuristic Solver work, and how networks of different shapes can be flattened in order to be solved by them, it's time to compare the performance of these solvers more thoroughly. In particular, we want to see how these solvers perform when the dial is turned on various parameters in randomly generated networks. Let's start with a basic network, and modify attributes from there:

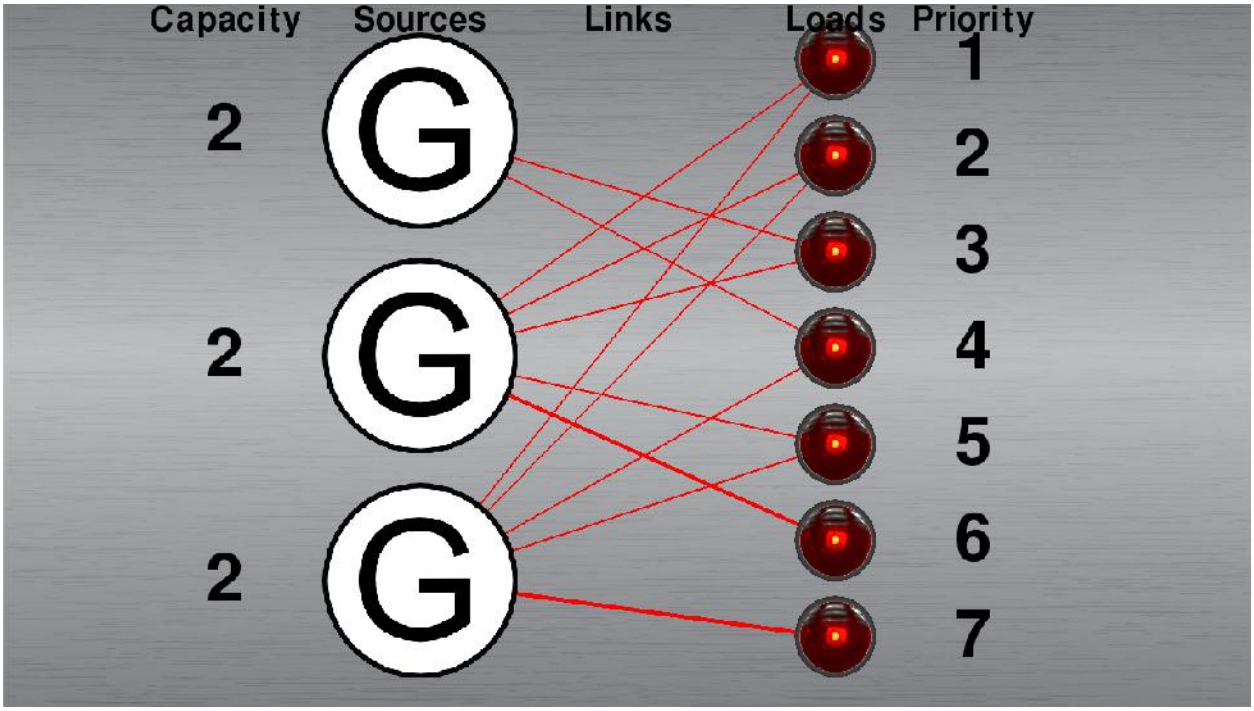

Figure 4.1: Random network.

This network is simple without being trivial. There are 7 loads and a total of 6 capacity to supply them, which means that not all of the loads can be supplied. There are 2 possible links by which each load can be supplied. There are 5 incompatibilities in the network, which, with only 14 links, is likely to affect the solution.

Here is the Heuristic Solution to this basic network:

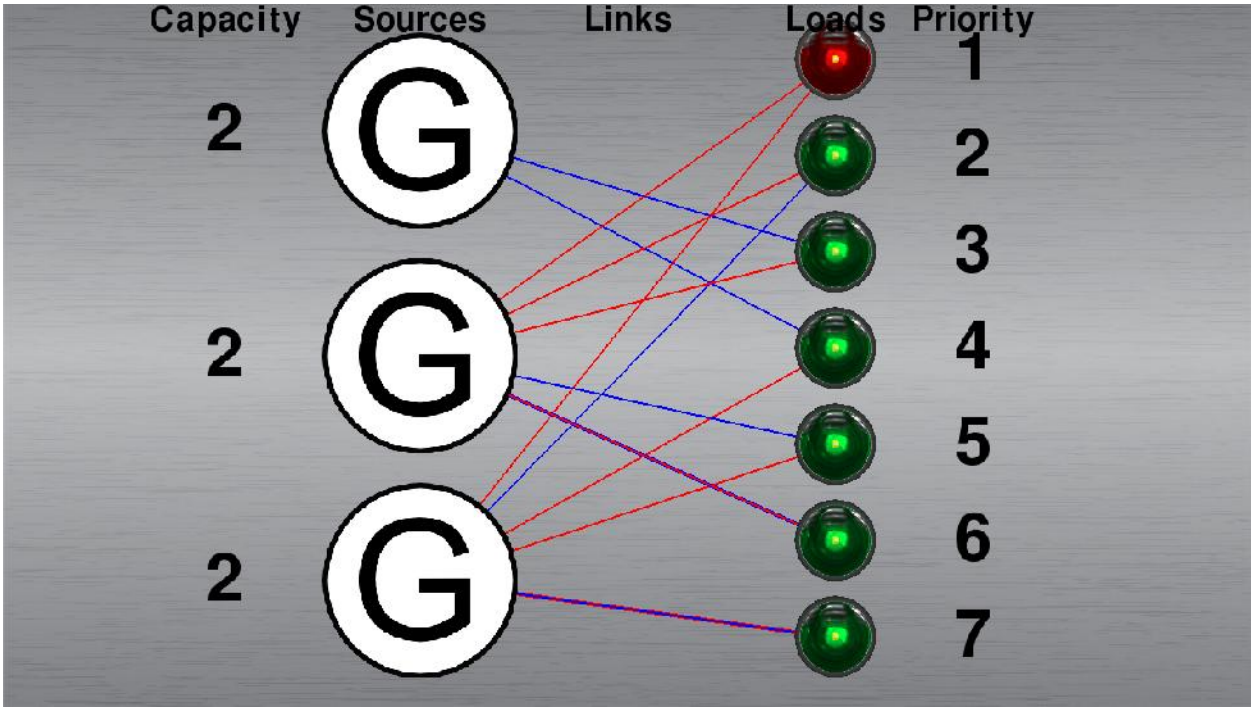

Figure 4.2: Random network, Heuristic Solution.

Here is the ILP Solution to the same network:

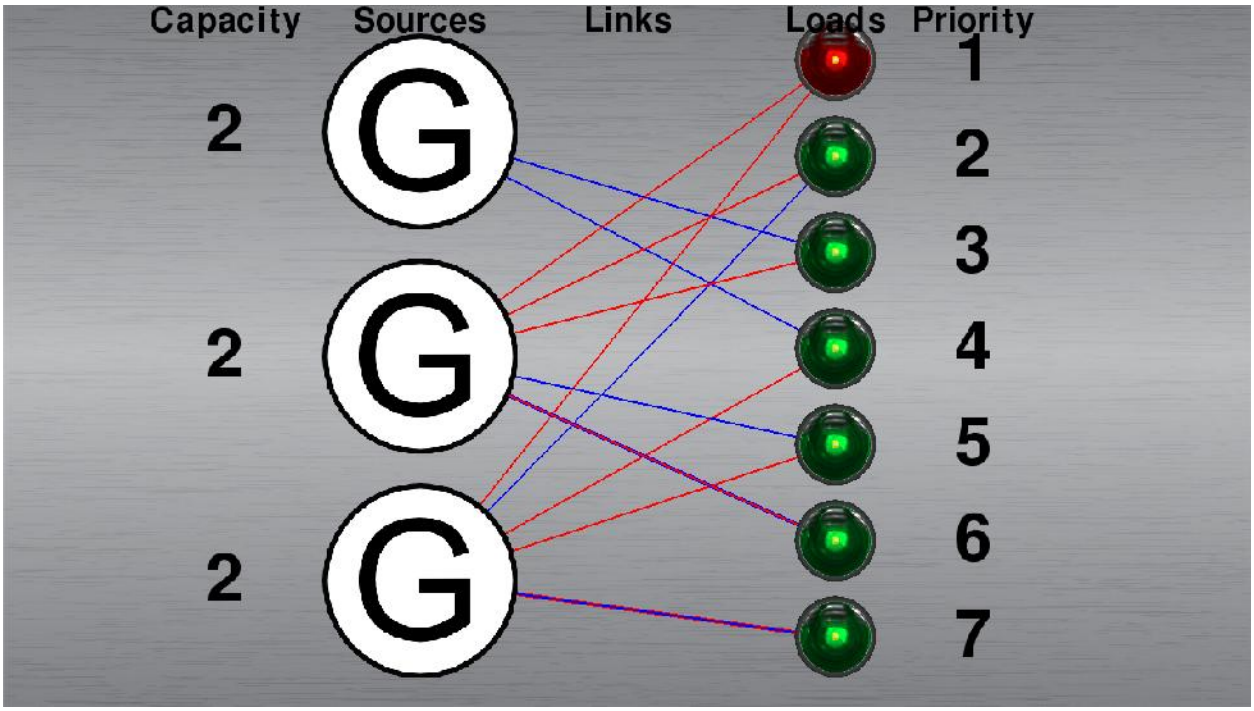

Figure 4.3: Random network, ILP Solution.

These solutions are actually identical. It took the Heuristic Solver 0.001 seconds to find it and the ILP Solver 1.248 seconds.

Here is the what happens when the number of incompatibilities in this network is steadily increased and the other parameters are kept the same:

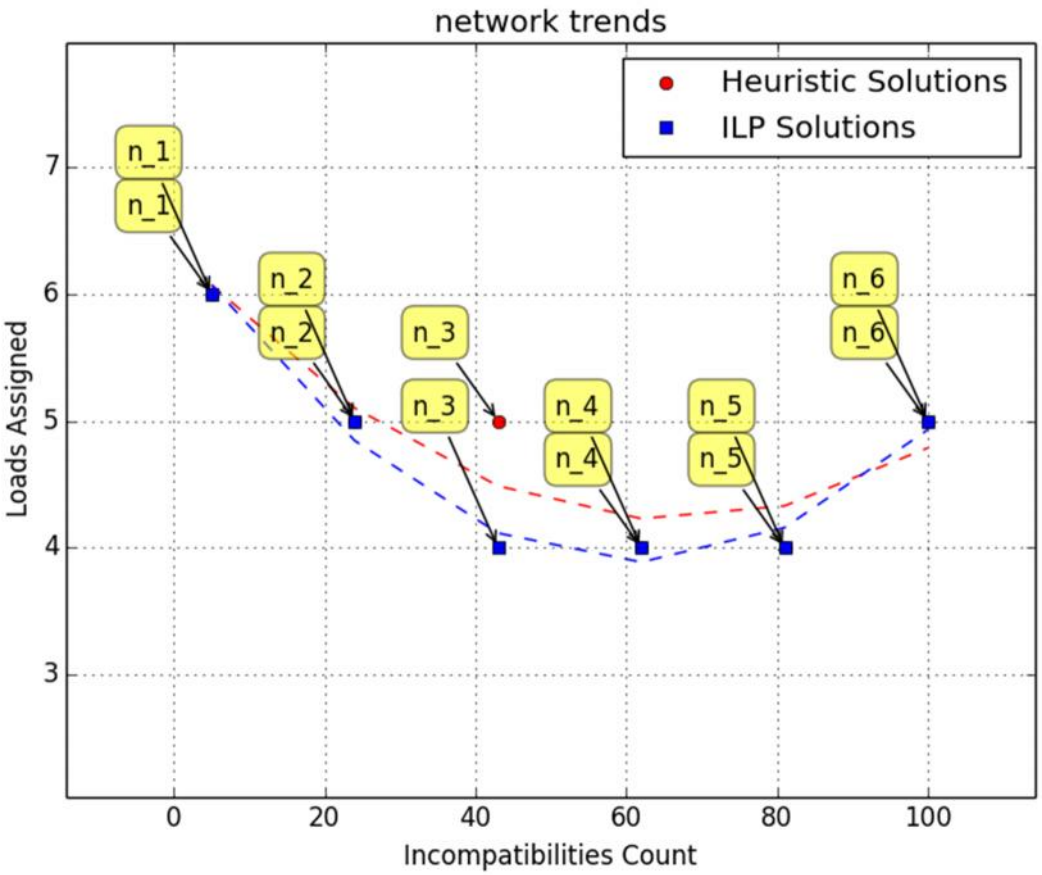

Figure 4.4: Loads assigned vs incompatibilities.

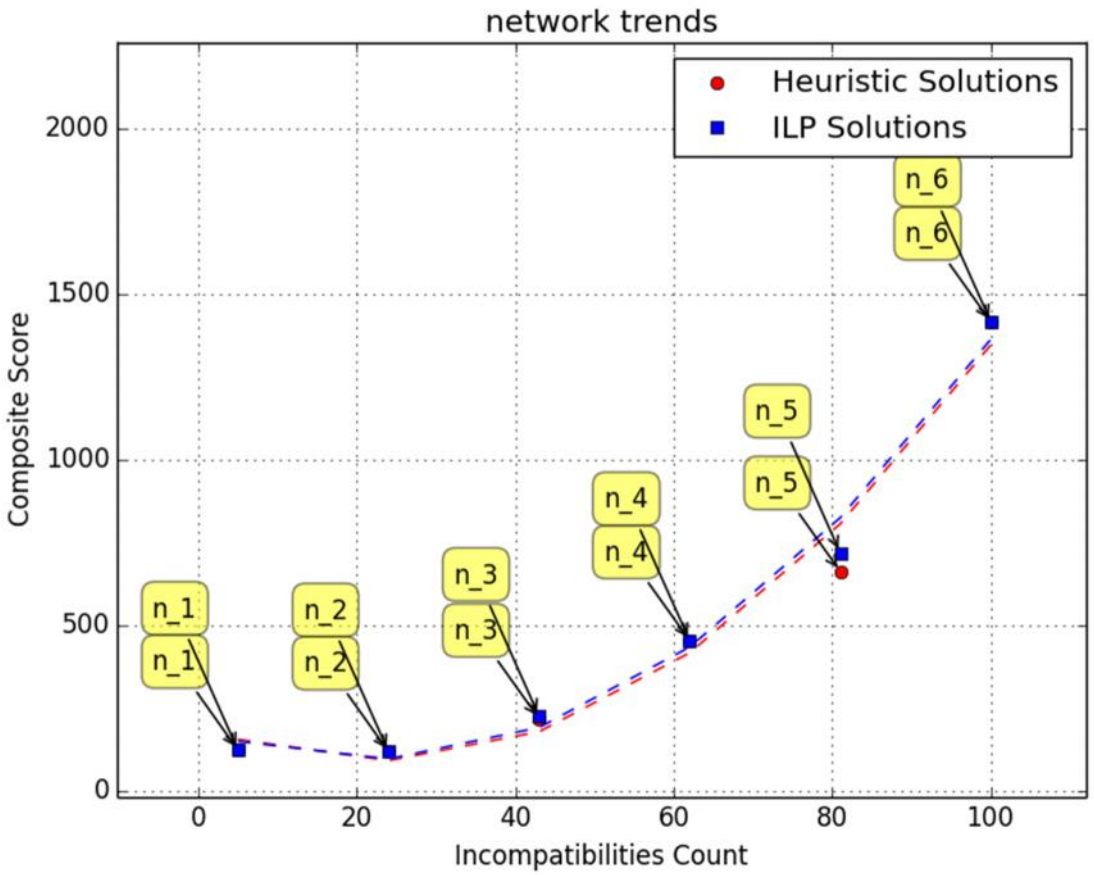

Figure 4.5: Composite score vs incompatibilities.

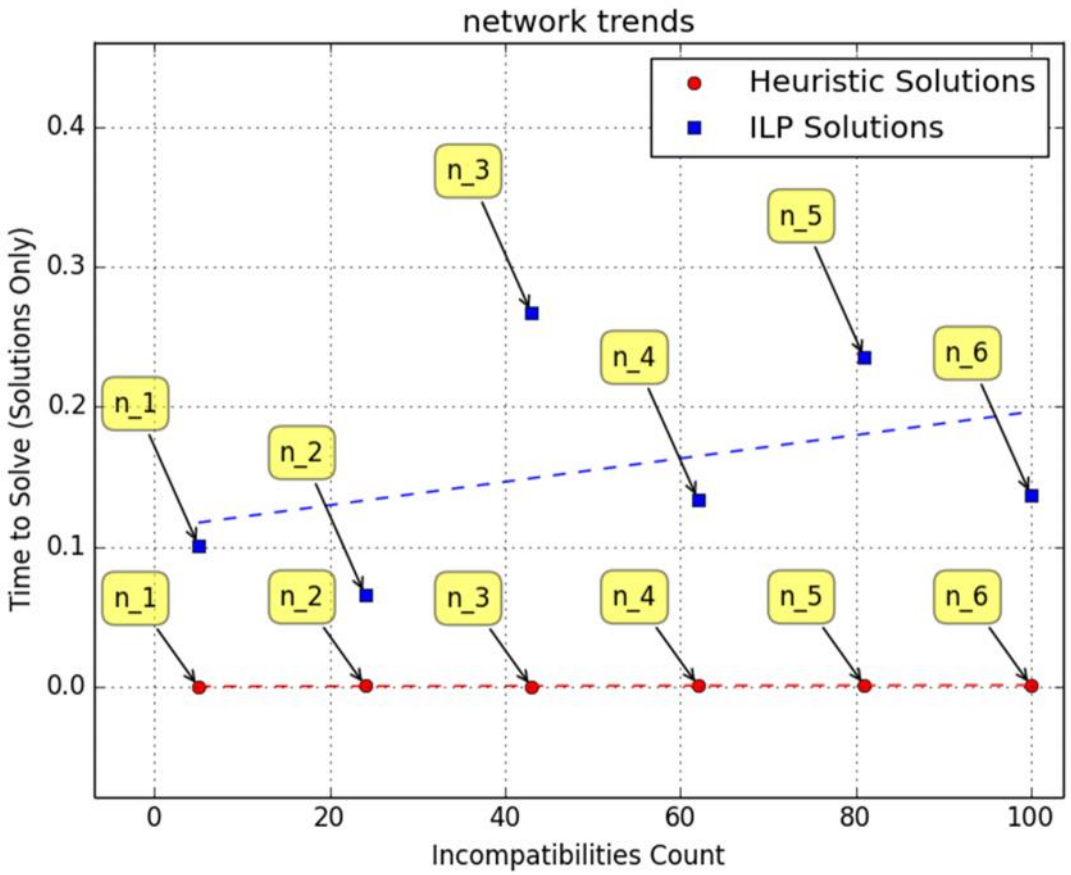

Figure 4.6: Time to solve vs incompatibilities.

These networks were all solved in batch, which seems to have improved their time to solve in the case of the ILP Solver, but the trend line still shows it increasing, whereas the Heuristic Solver remains incredibly fast.

We have to zoom in very close to even see the time it took for the Heuristic Solver to register on the chart at all:

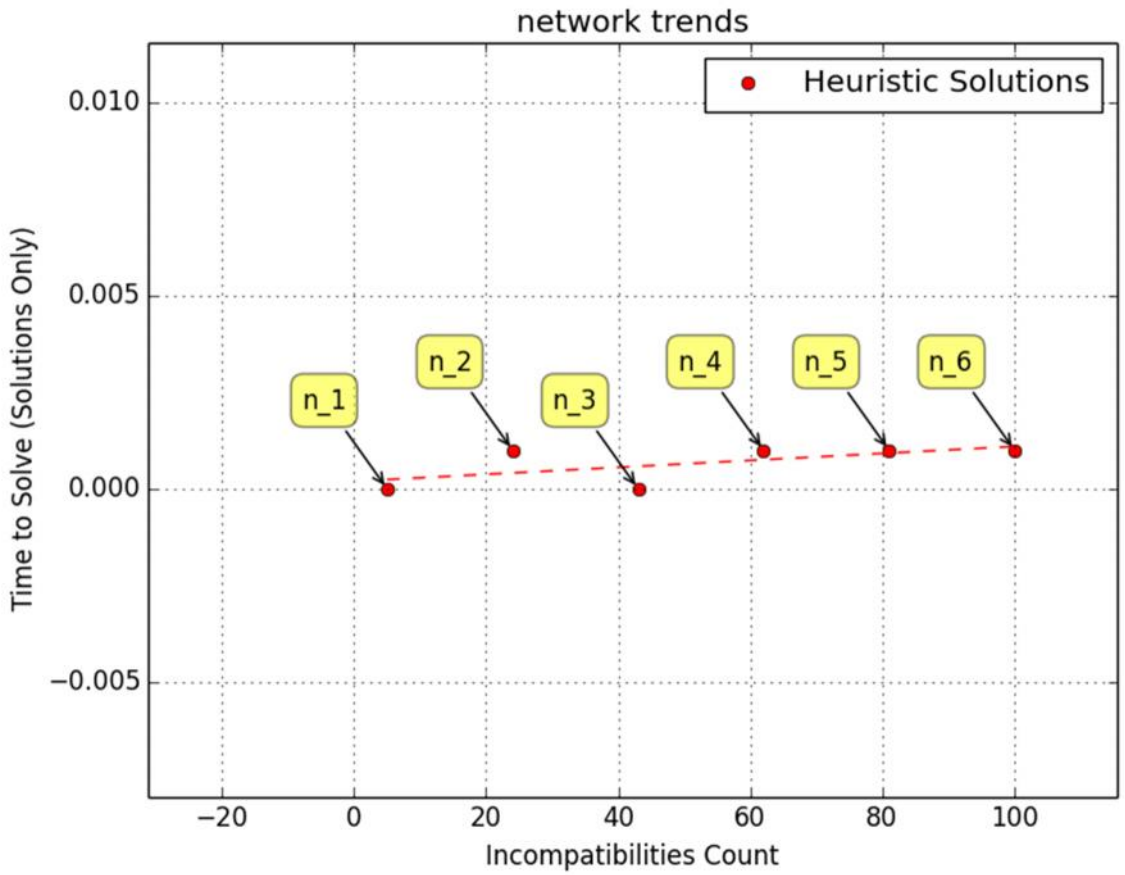

Figure 4.7: Time to solve vs incompatibilities, Heuristic Solutions.

Let's see what happens when we make steadily increase the complexity of the network altogether. With each step, there is an even progression from the original source count, load count, and incompatibilities count of 3, 7, and 5 to 20, 70, and 500, respectively. At each step, we randomly generate 3 networks with those given parameters.

Here are the number of loads assigned:

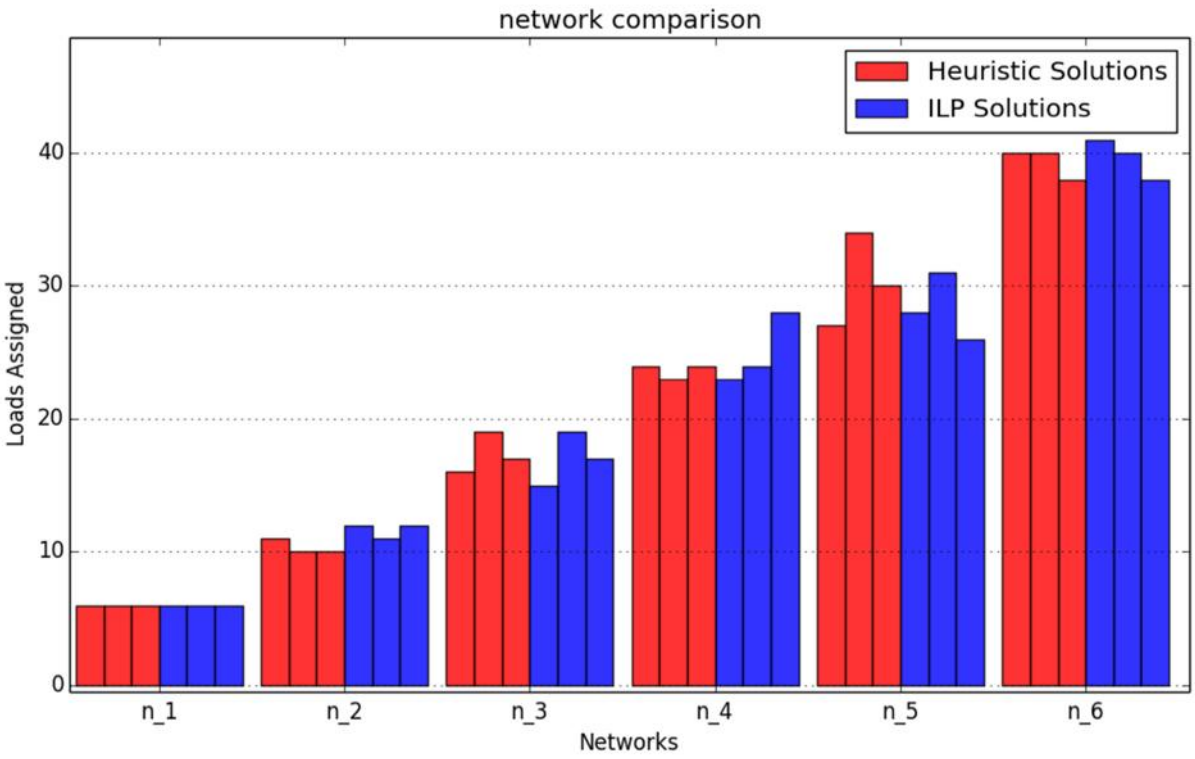

Figure 4.8: Loads assigned with increasing network complexity.

The composite scores for each of these same networks are difficult to plot in one chart because as the number of loads increases, the total possible composite score increases exponentially. Here are each set of three networks with the same parameters at each step:

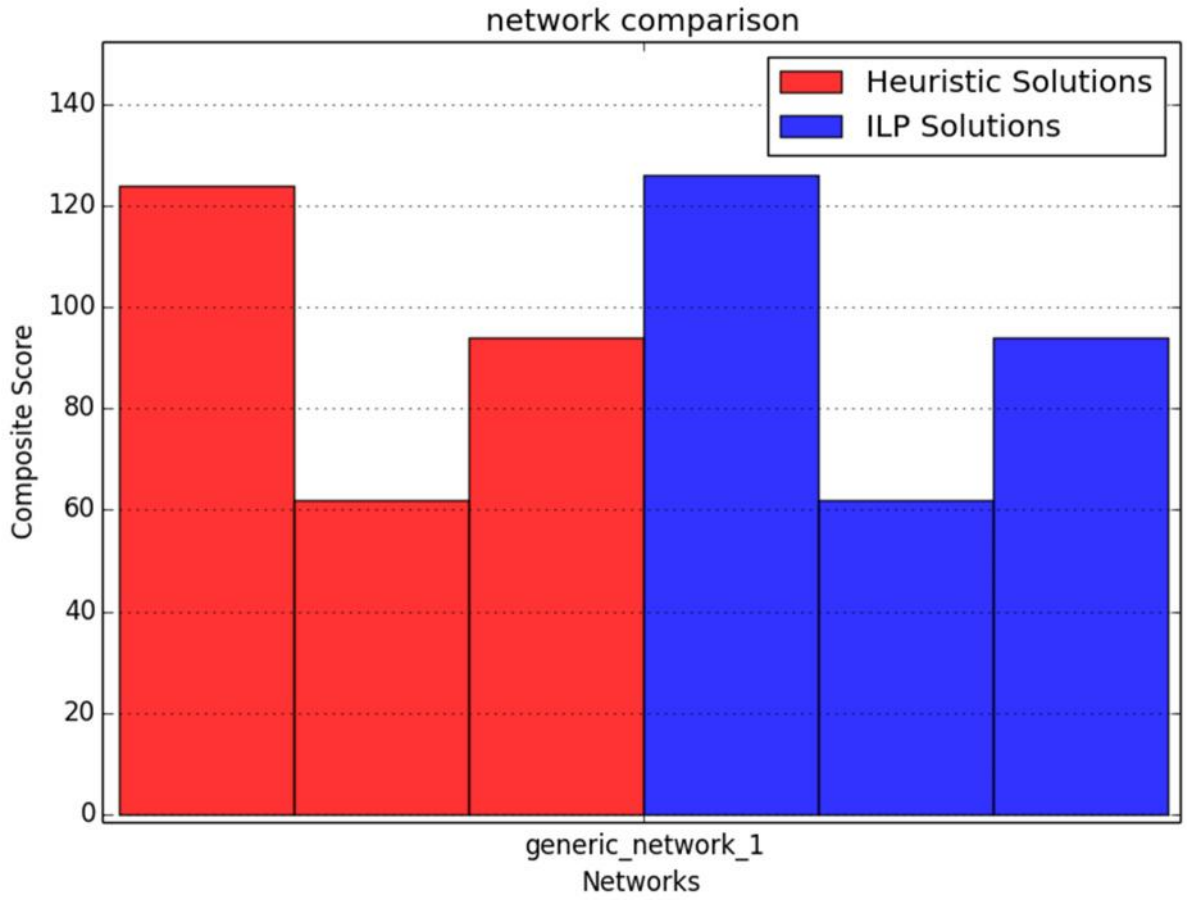

Figure 4.9: Composite score with increasing network complexity 1.

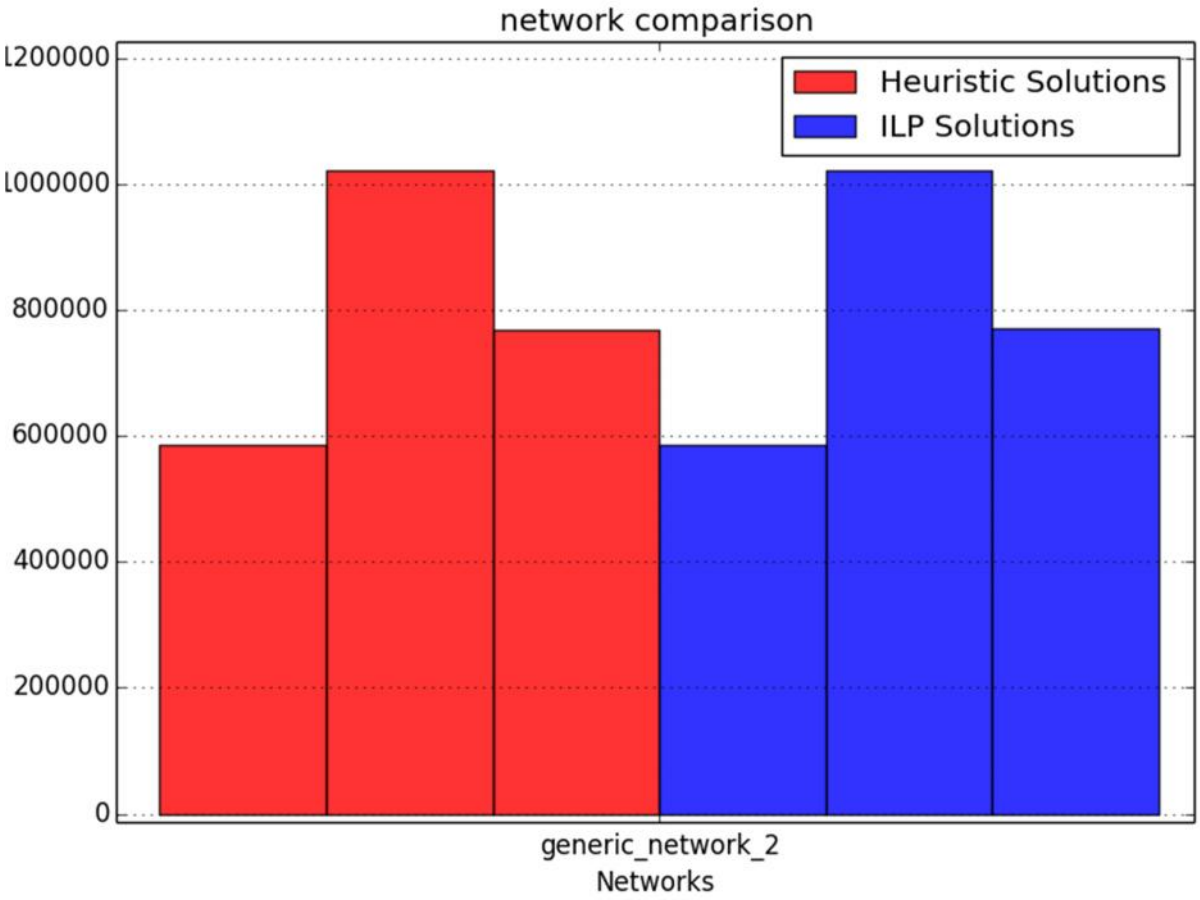

Figure 4.10: Composite score with increasing network complexity 2.

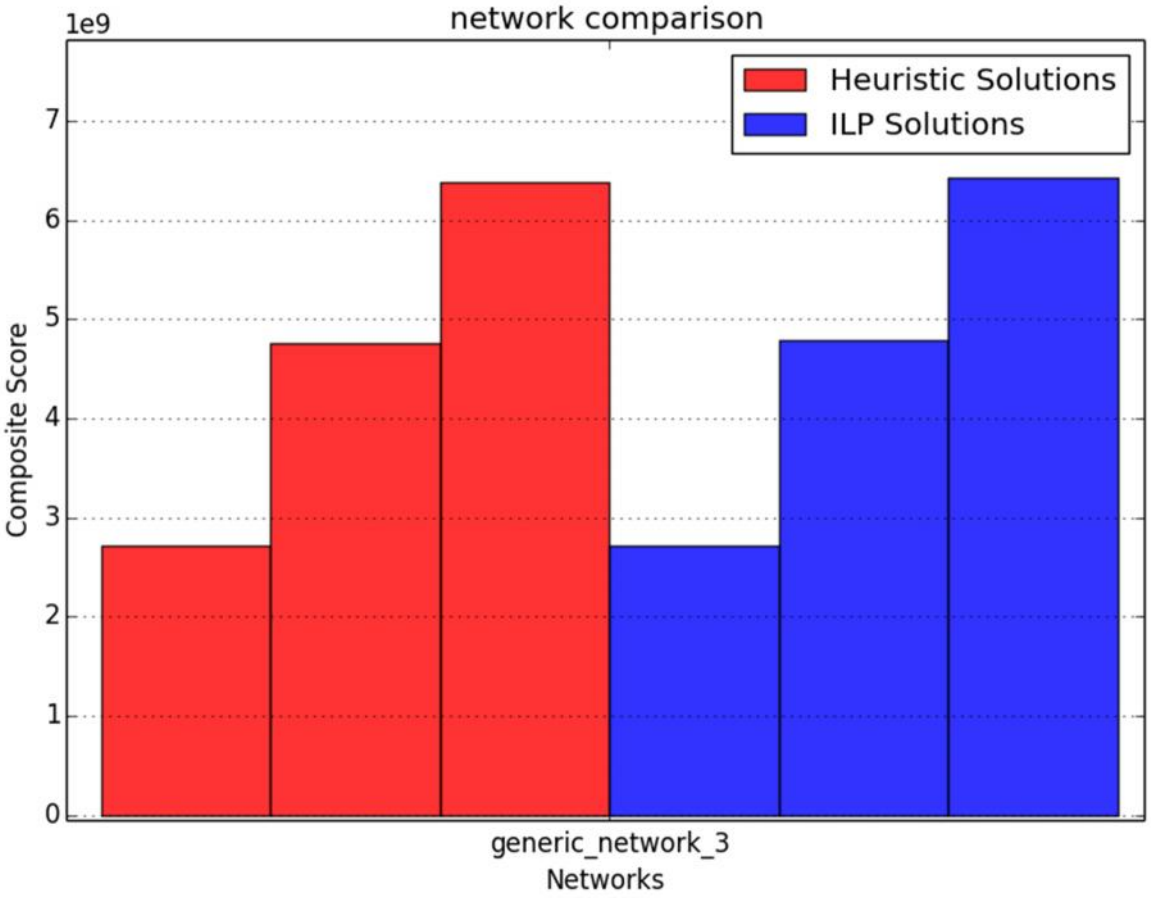

Figure 4.11: Composite score with increasing network complexity 3.

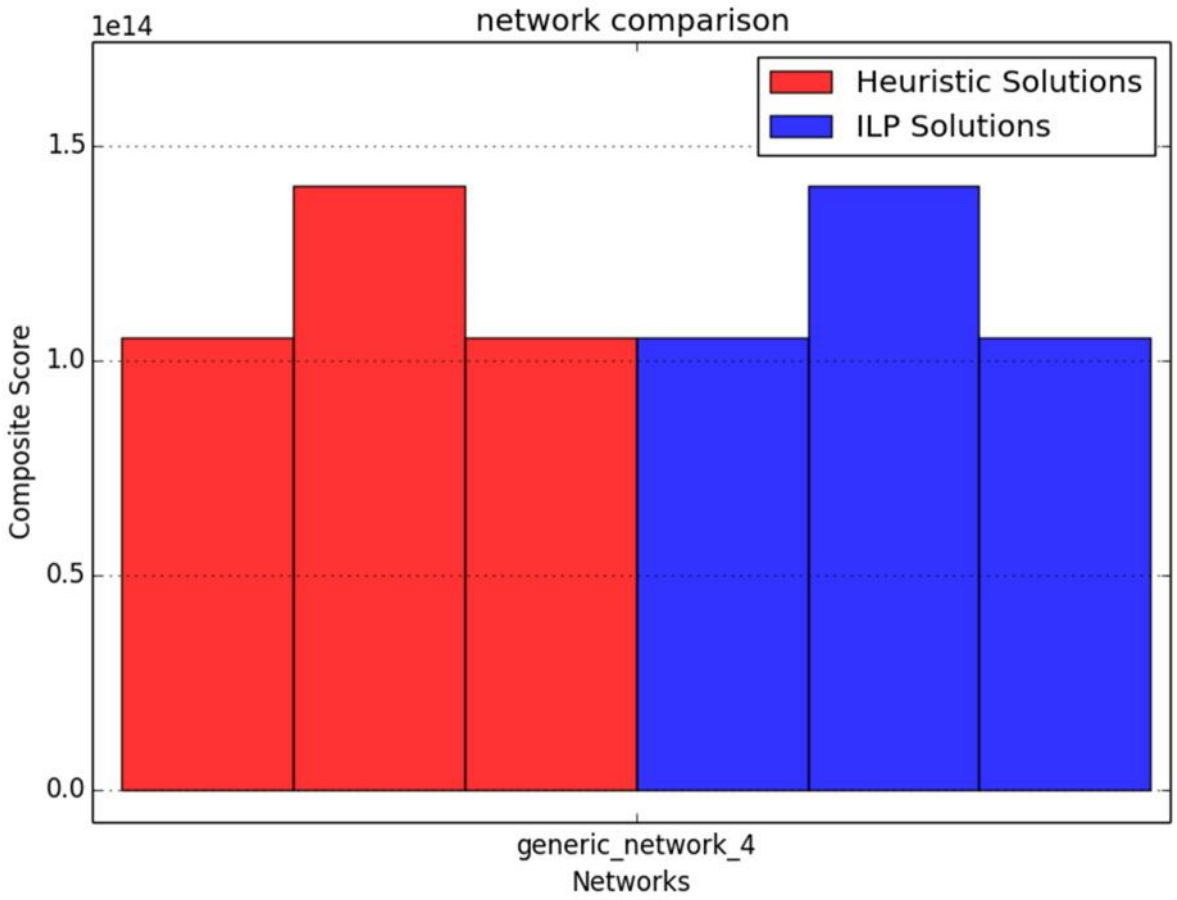

Figure 4.12: Composite score with increasing network complexity 4.

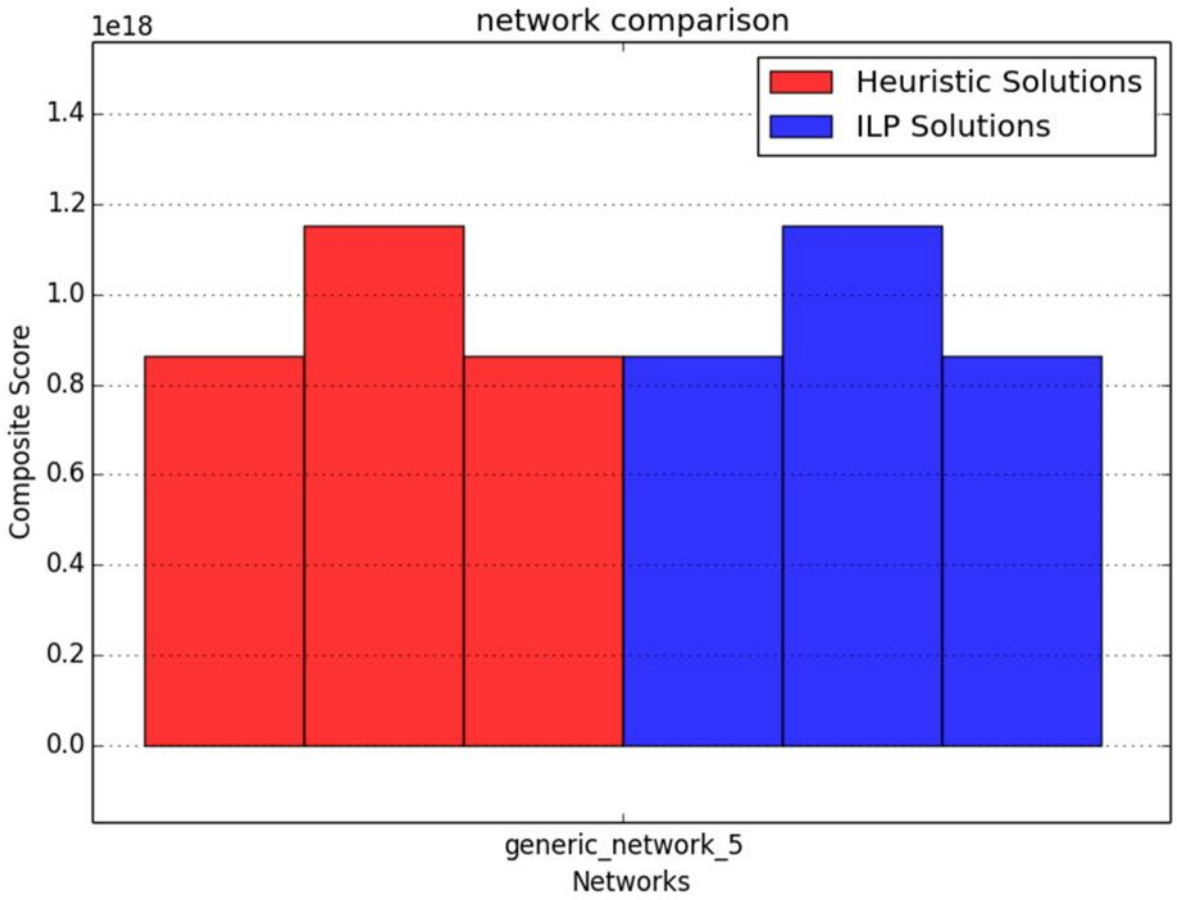

Figure 4.13: Composite score with increasing network complexity 5.

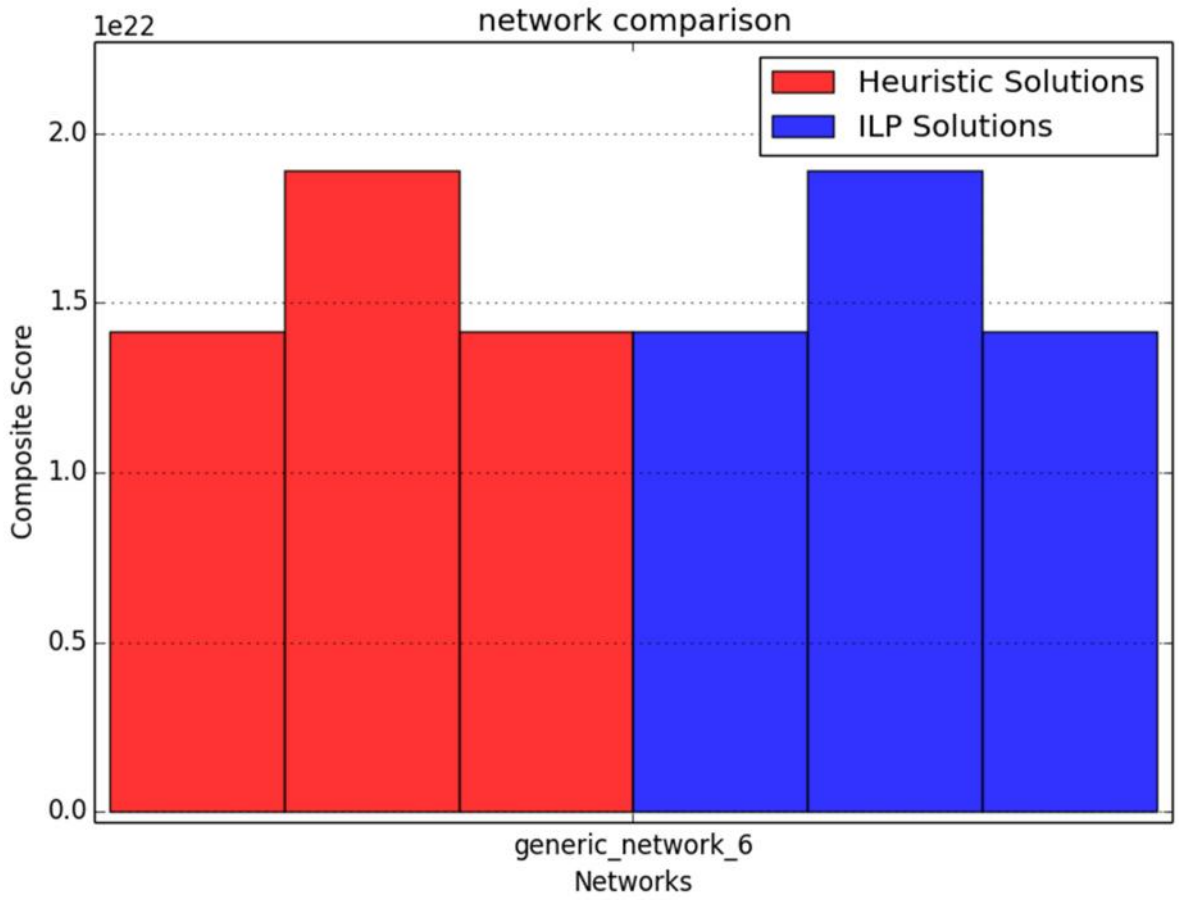

Figure 4.14: Composite score with increasing network complexity 6.

The trend seems to be that the more complex a network, the smaller the gap is between the ILP Solution's composite score and the Heuristic Solution's composite score, but it is almost impossible to see the difference between the scores at any of these steps. The time to solve each, however, is not even close to similar:

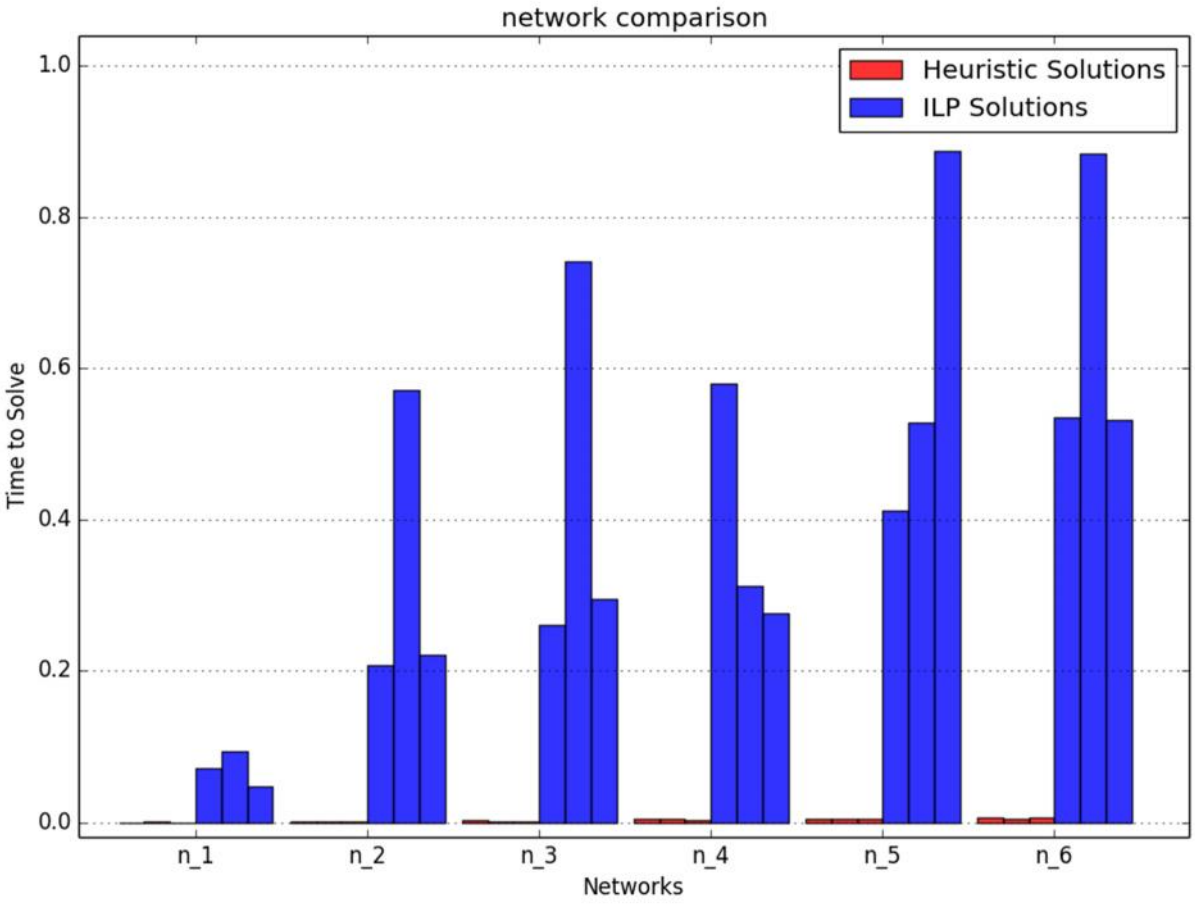

Figure 4.15: Time to solve with increasing network complexity.

It is difficult to see the Heuristic Solver's time to solve at each step because of how small it is in comparison to the ILP Solver's time to solve. It is easy to see how the ILP Solver can very quickly become impractical or infeasible to wait on, especially for minimal gains in the quality of the solution computed.

To further bolster this point, let's revisit the synthetic network generated in the previous section, Flattening the Distribution Network. Here is the Heuristic Solution for this network:

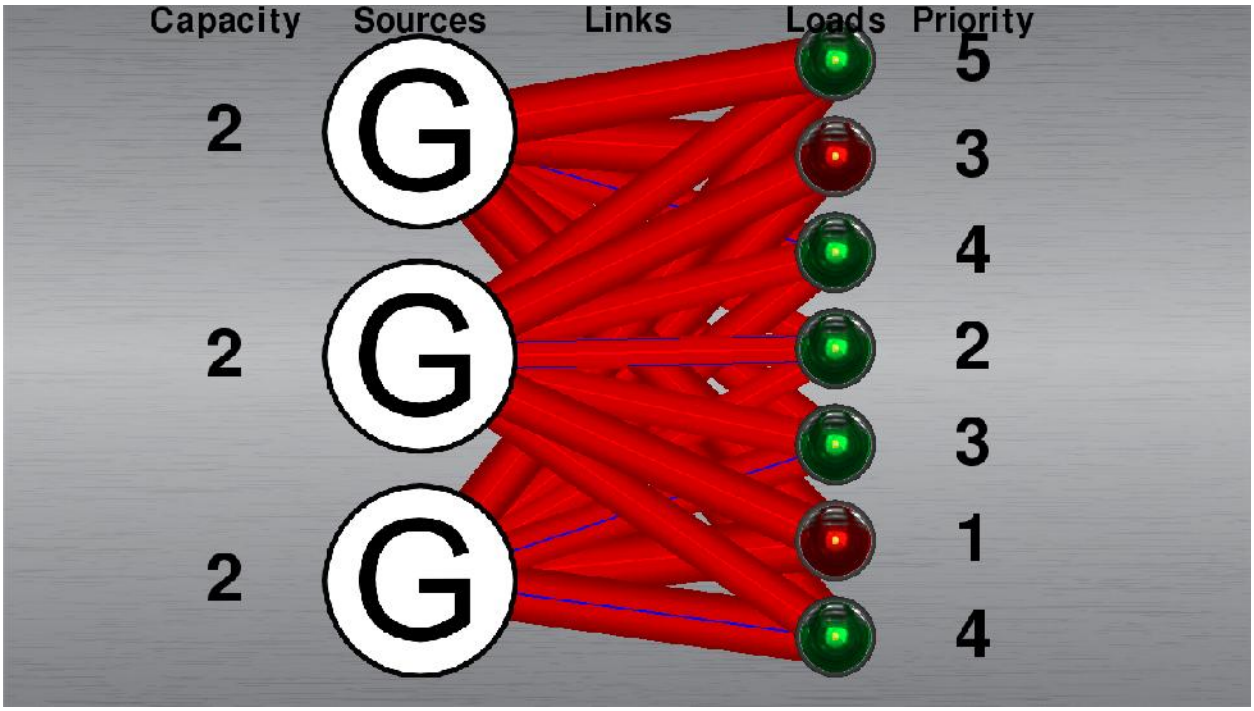

Figure 4.16: Synthetic Network, Heuristic Solution.

Here is the ILP Solution for the same network:

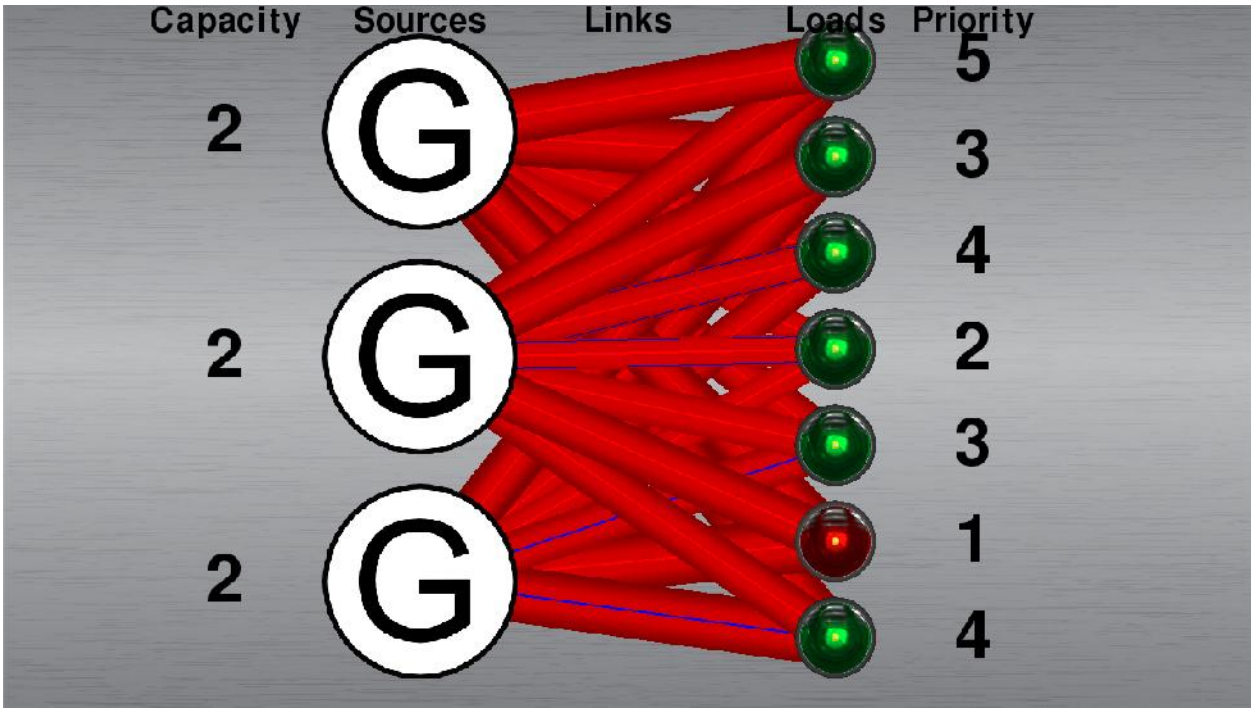

Figure 4.17: Synthetic Network, ILP Solution.

As you can see, this is an example where the Heuristic Solver returns a suboptimal solution:

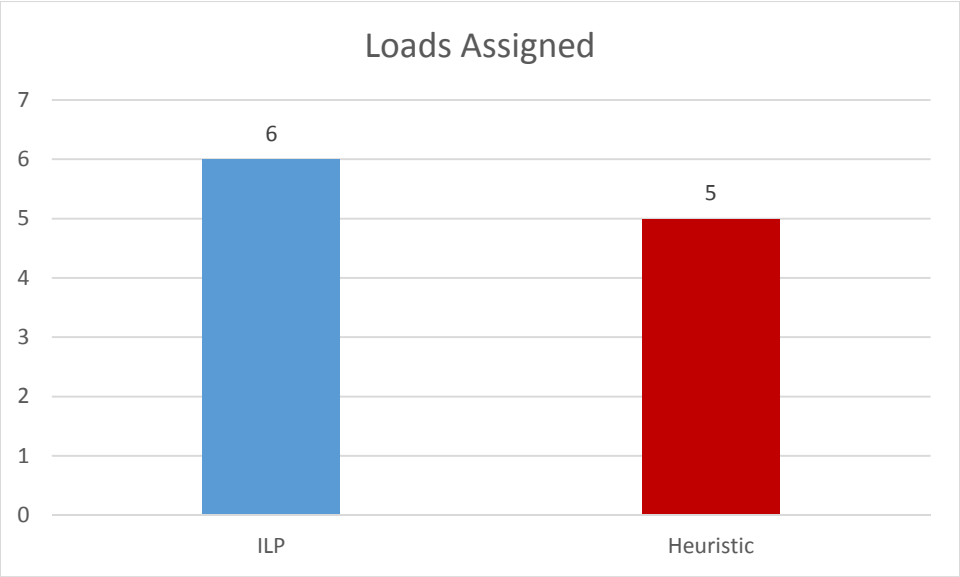

Figure 4.18: Synthetic Network loads assigned.

This might seem concerning, until two things are considered. First is a comparison of the quality of those solutions:

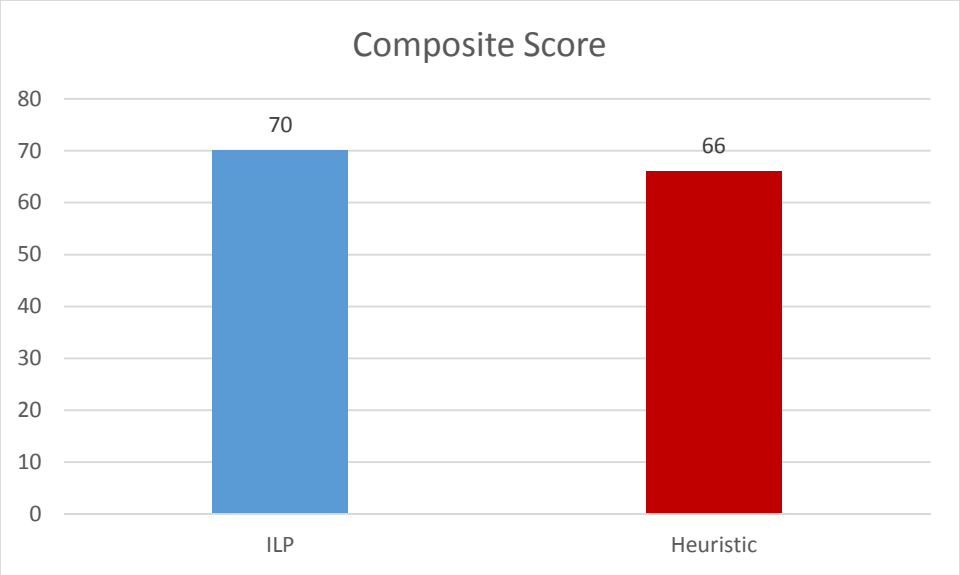

Figure 4.19: Synthetic Network composite score.

The Heuristic Solver achieves a 94.3% perfect solution, which is likely to be as good a solution as can be hoped for with this particular network in any reasonable amount of time:

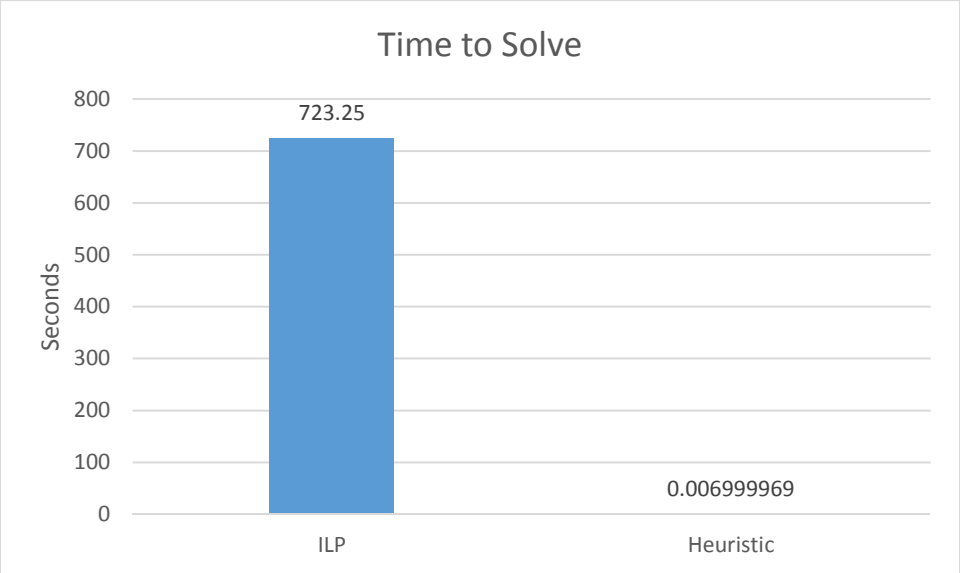

Figure 4.20: Synthetic Network time to solve.

It is important to emphasize that the ILP Solver took 12 minutes and 3.25 seconds to finish, while the Heuristic Solver took 7 milliseconds to find a solution for the same network. This is over 100,000 times longer! The likely cause of this is because while there were only 3 sources and 7 loads in this network, once the topology was flattened there were 511 links with a sum total of 86,031 links to keep track of, which is very many constraints for a linear programming problem!

It should be noted that testing solutions on the Synthetic Network is what caused the rule of checking links with fewest incompatibilities first—there are many ways for pathways to cut each other off in a grid structure. This is the same reason why the Heuristic Solver left out a load in its solution, but also why this network is a good test.

Another good test of these solvers is the Benchmark Network, which was also flattened in the previous section. Here is the result from the Heuristic Solver:

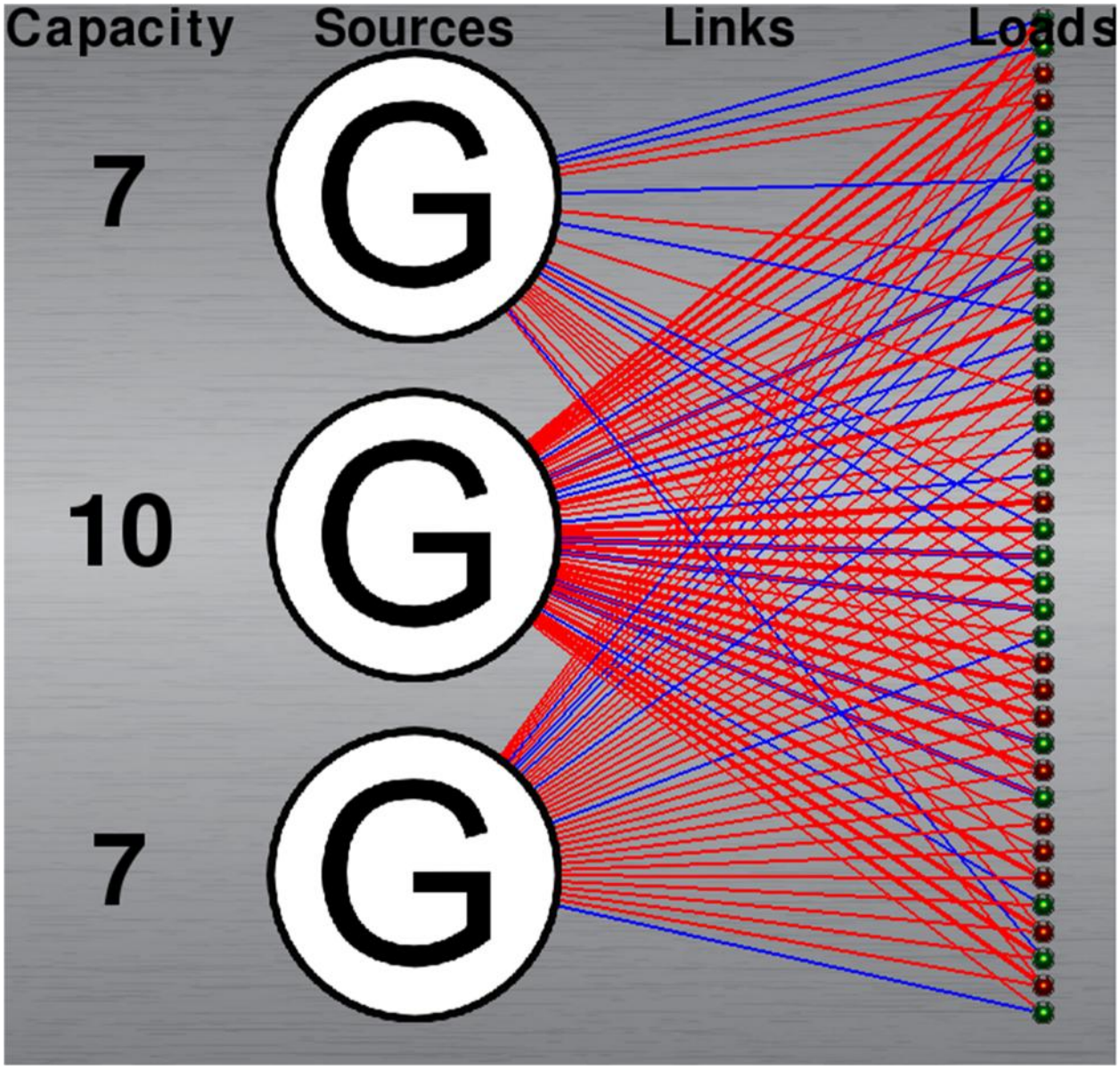

Figure 4.21: Benchmark Network, Heuristic Solution.

Here is the result from the ILP Solver:

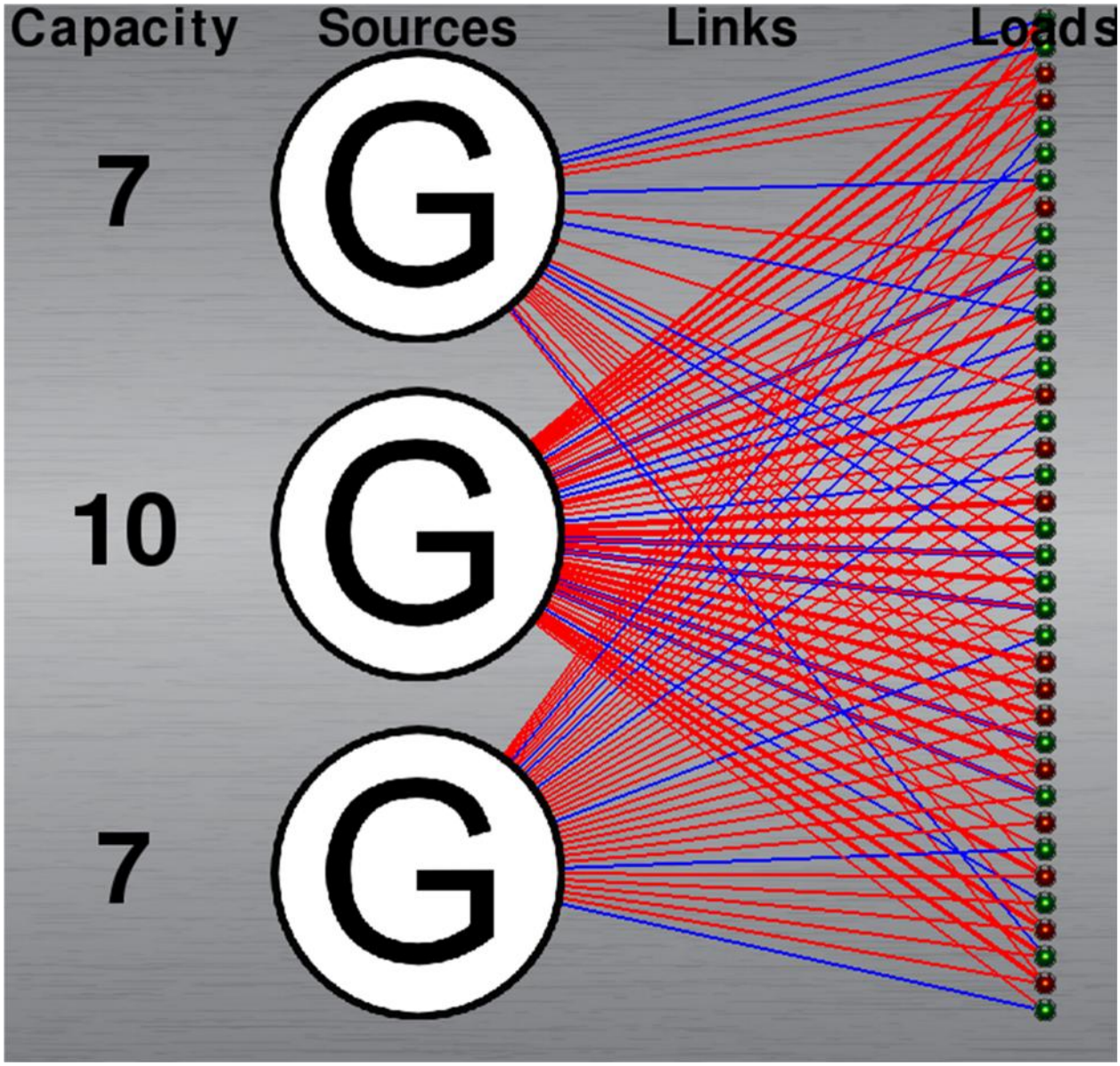

Figure 4.22: Benchmark Network, ILP Solution.

The results may be easier to see quantitatively:

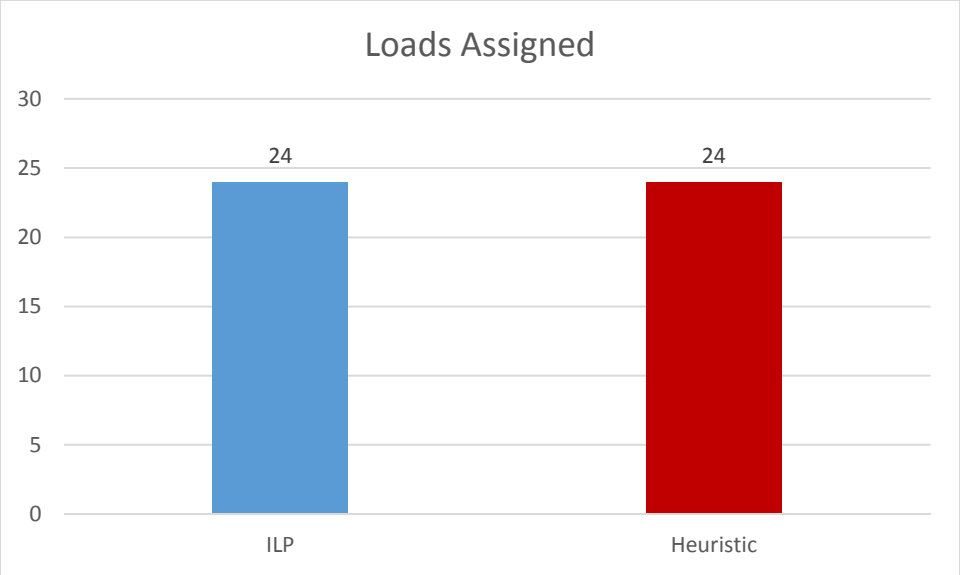

Figure 4.23: Benchmark Network loads assigned.

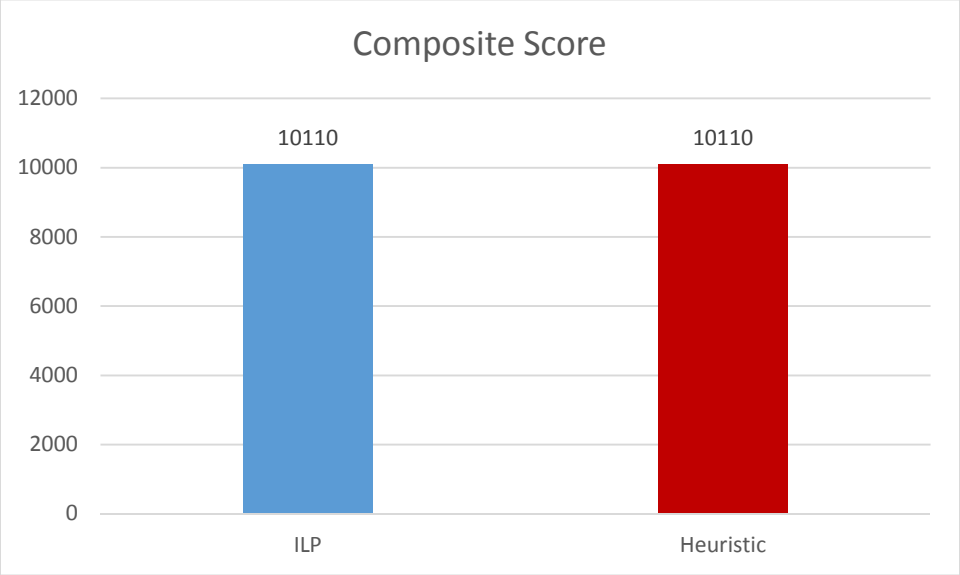

Figure 4.24: Benchmark Network composite score.

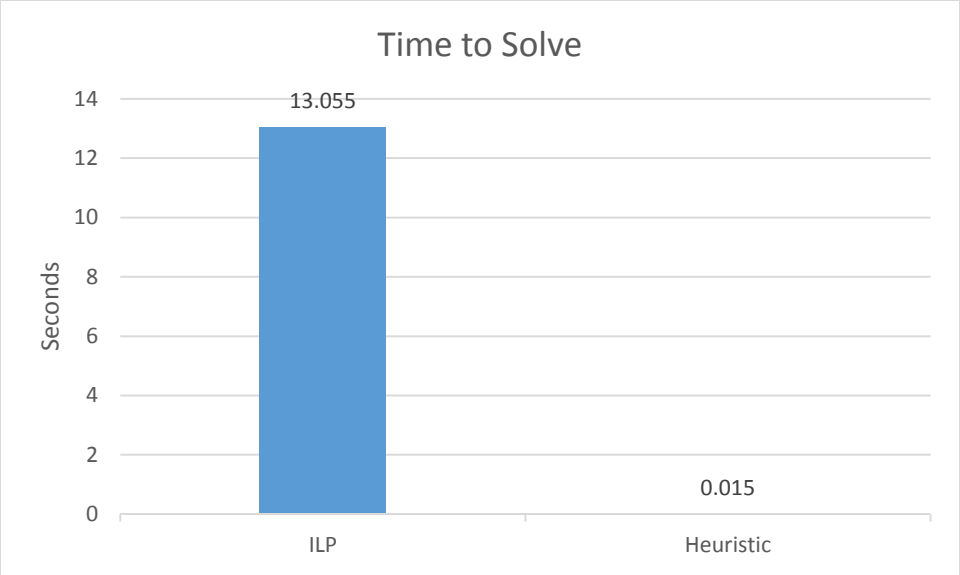

Figure 4.25: Benchmark Network time to solve.

The Heuristic Solver finds the objectively best solution for the Benchmark Network 870.3 times faster than the ILP Solver. It should be noticed that this was the circuit that caused the rule about sending "choking sources" to the bottom of the candidate link order to be introduced, as when incompatibilities only are considered, G3 very quickly gets cut off as loads near it favor paths from G1.

To be sure that these results for the Benchmark Network are not an outlier, these solvers were run again with different priority levels assigned to the same groups of loads, first ranking them by number of users, then by category:

Group	Loads	Number of Customers	Priority (Trial 2)
A	L1, L2, L3, L4, L11, L12, L13, L18, L19, L20, L21, L32, L33, L34, L35	220	4

B	L5, L14, L15, L22, L23, L36, L37	200	3
C	L8, L10, L26, L27, L28, L29, L30	1	1
D	L9, L31	1	1
E	L6, L7, L16, L17, L24, L25, L38	10	2

Table 4.1: Benchmark Network load priorities 2.

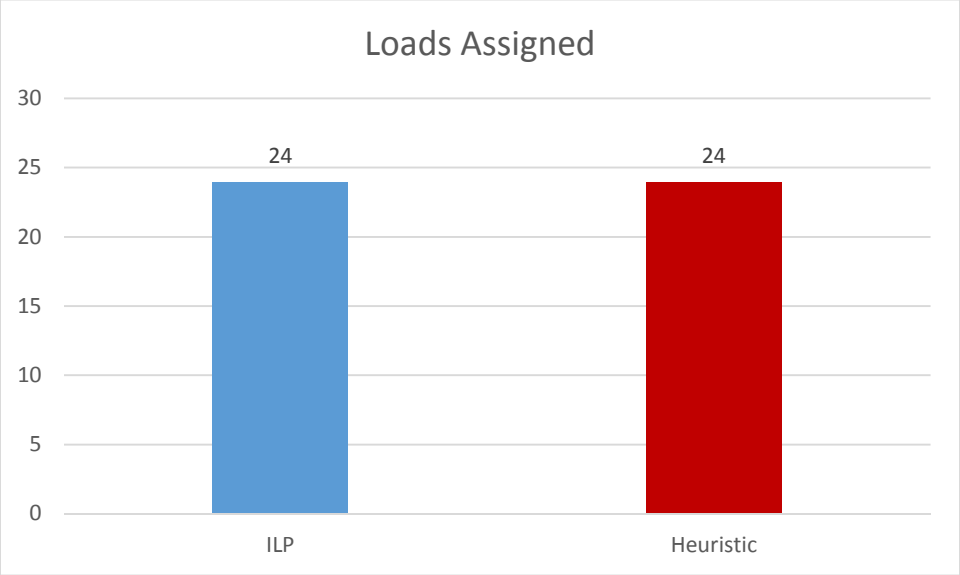

Figure 4.26: Benchmark Network loads assigned 2.

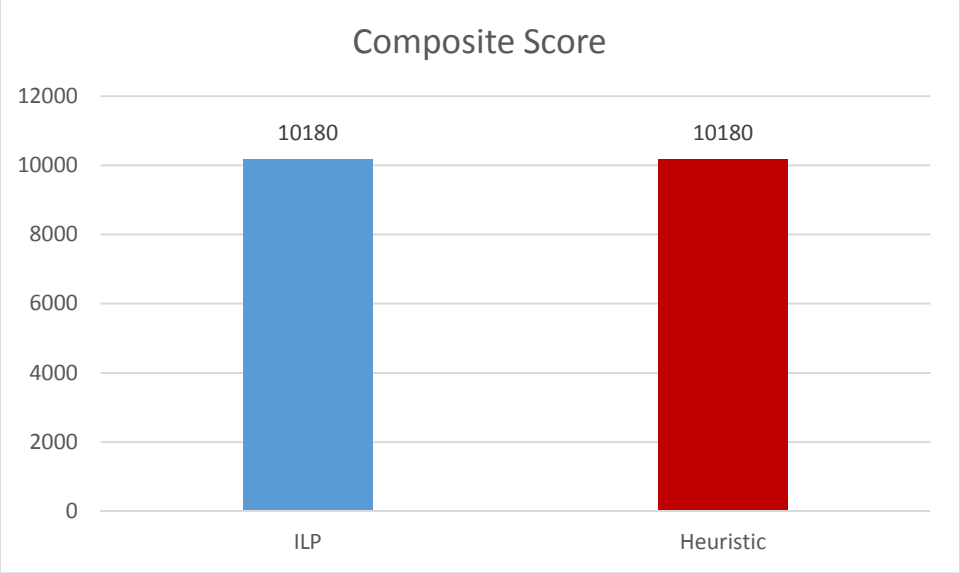

Figure 4.27: Benchmark Network composite score 2.

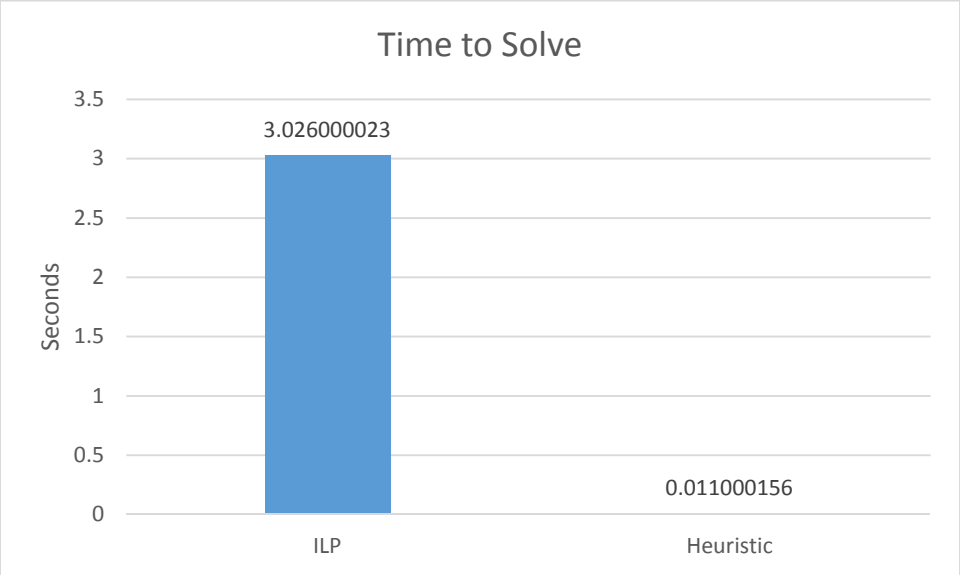

Figure 4.28: Benchmark Network time to solve 2.

Group	Loads	Customer Type	Priority (Trial 3)
A	L1, L2, L3, L4, L11, L12, L13, L18, L19, L20, L21, L32, L33, L34, L35	residential	2
B	L5, L14, L15, L22, L23, L36, L37	residential	2
C	L8, L10, L26, L27, L28, L29, L30	small user	1
D	L9, L31	small user	1
E	L6, L7, L16, L17, L24, L25, L38	commercial	3

Table 4.2: Benchmark Network load priorities 3.

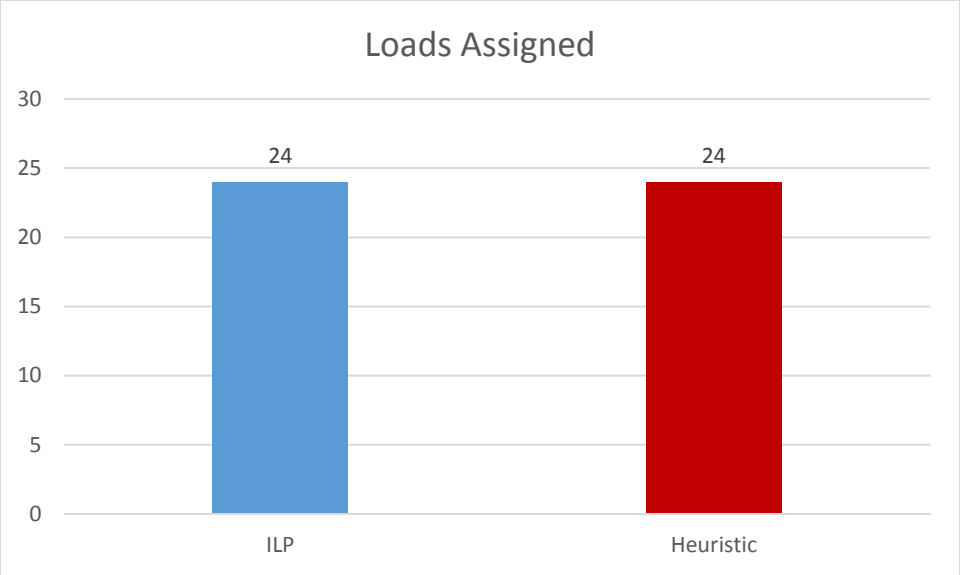

Figure 4.29: Benchmark Network loads assigned 3.

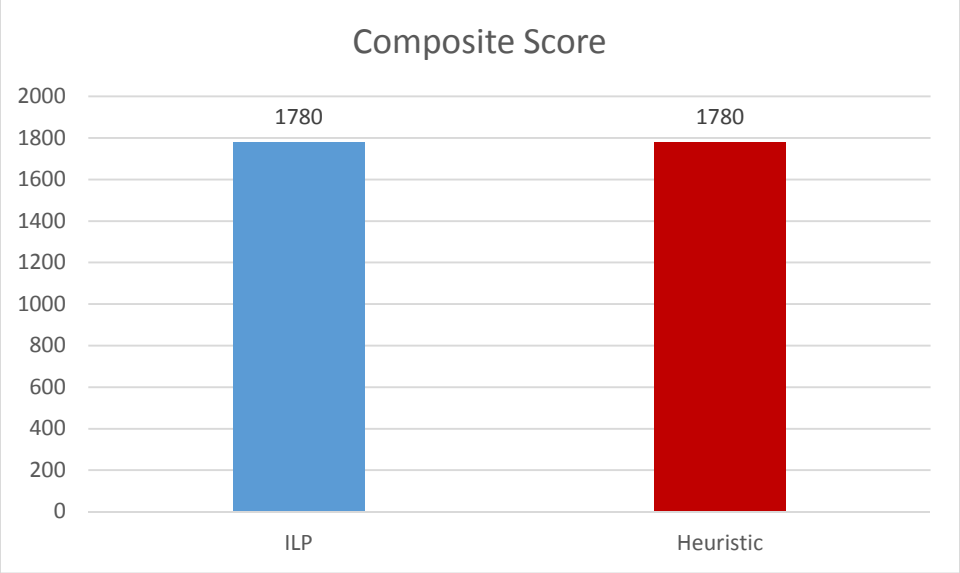

Figure 4.30: Benchmark Network composite score 3.

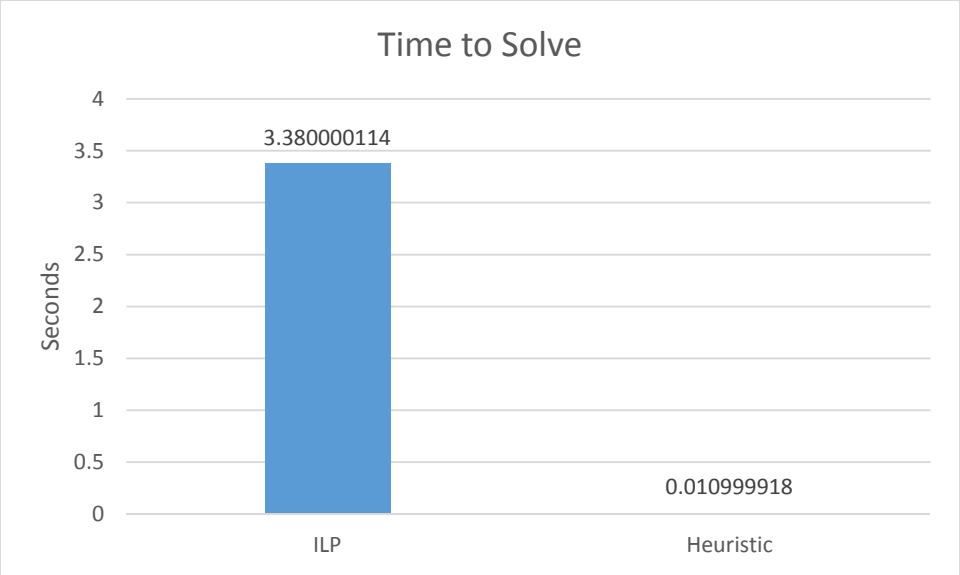

Figure 4.31: Benchmark Network time to solve 3.

As these trials show, the Heuristic Solver continues to return an optimal solution for the Benchmark Network at a small fraction of the time it takes for the ILP Solver to find a solution of equal quality. These kinds of results are promising and necessary for any power distribution network reconfiguration algorithm to be used in sensitive applications.

CONCLUSION & FUTURE WORK

We have proposed a method by which the topology of a network might be discovered through an algorithm like the distributed Bellman-Ford algorithm. We have explored the inner workings of two methods to automate power distribution network reconfiguration, the ILP Solver and the Heuristic Solver. We have seen how networks of different shapes can be translated into a flattened topology, which is necessary preprocessing to find a power assignment solution for a network. We have also seen some experimental results comparing the performance of the ILP Solver and the Heuristic Solver.

The Heuristic Solver is a very fast, efficient algorithm to reconfigure power distribution, which is important in the case of an emergency. It performs consistently with near perfect results at a speed that is orders of magnitude quicker than the ILP Solver in almost all cases. In an application where a network is small and time is not an important constraint, the ILP Solver could possibly be preferable, but in any context where time is sensitive and near-perfect results are as acceptable as perfect results, the Heuristic Solver is much preferable.

There is always room for improvement. Future tests should perhaps allow for non-integer capacity units, or loads that require other values than unit capacity. Optimizing each algorithm by rewriting them in C could give more optimized tests, though this may not be necessary to make judgments about implementing one or the other. There may be some ways to improve the Heuristic Solver, such as arranging the *ordered_links* in some way that could be more optimal. The algorithm could also be improved by taking advantage of the fact that once there are no more sources with capacity to provide any loads, the process of trying to assign loads to them for

power supply can cease. Perhaps this method could be combined with other methods that do not presently account for load priorities or place as much value on fast execution.

REFERENCE

[1] R. Rao, K. Ravindra, K. Satish and S. Narasimham, "Power Loss Minimization in Distribution System Using Network Reconfiguration in the Presence of Distributed Generation", *IEEE Transactions on Power Systems*, vol. 28, no. 1, pp. 317-325, 2013.

[2] K. Hedman, S. Oren and R. O'Neill, "A review of transmission switching and network topology optimization", *2011 IEEE Power and Energy Society General Meeting*, p.p. 1-7, 2011.

[3] D. Haughton and G. Heydt, "Smart distribution system design: Automatic reconfiguration for improved reliability", *IEEE PES General Meeting*, pp. 1-8, 2010.

[4] M. Farzadi, F. Heris, F. Shahir and A. Sedighmanesh, "The role of the intelligent reconfiguration of distribution network on reduction the energy not supplied costs in the electricity market through case studies using softwares NEPLAN & DIgSILENT Power Factory", *2015 9th International Conference on Electrical and Electronics Engineering (ELECO)*, 2015.

[5] T. Thakur and J. Dhiman, "Study and characterization of power distribution network reconfiguration", *2006 IEEE/PES Transmission & Distribution Conference and Exposition: Latin America*, pp. 1-6, 2006.

[6] B. Vyas, M. Sharma and S. Jain, "Feeder Reconfiguration of distribution network using Minimum Power Flow methodology", *2015 Annual IEEE India Conference (INDICON)*, 2015.

[7] Li Ouyang, Yong Li, Yi Tan, Juanxia Xiao and Yijia Cao, "Reconfiguration optimization of DC zonal distribution network of shipboard power system", *2016 IEEE Transportation Electrification Conference and Expo, Asia-Pacific (ITEC Asia-Pacific)*, 2016.

[8] M. Jünger, *50 years of integer programming 1958-2008*. Berlin: Springer, 2010.

[9] D. Bertsekas and R. Gallager, *Data networks*, 2nd ed. Englewood Cliffs, N.J.: Prentice Hall, 1992, pp. 404-410.

[10] S. Bradley, A. Hax and T. Magnanti, *Applied mathematical programming*. Reading, Mass.: Addison-Wesley Pub. Co., 1977.

[11] R. Allan, R. Billinton, I. Sjarief, L. Goel and K. So, "A reliability test system for educational purposes-basic distribution system data and results", *IEEE Transactions on Power Systems*, vol. 6, no. 2, pp. 813-820, 1991.

[12] A. Kouzou and R. Mohammedi, "Optimal reconfiguration of a radial power distribution network based on Meta-heuristic optimization algorithms", *2015 4th International Conference on Electric Power and Energy Conversion Systems (EPECS)*, 2015.

VITA

Graduate School

Southern Illinois University

Aaron J. Ekstrand

aaronekstrand@gmail.com

Cornell College

Bachelor of Arts, Computer Science and English, May 2013

Thesis Paper Title:

A Fast and Efficient Method for Power Distribution Network Reconfiguration

Major Professor: Dimitri Kagaris

Publications:

C. Loh and A. Ekstrand, “Audialization in serious games analytics: visualizing player performance improvement by sound or music”, *3rd International Workshop on Intelligent Digital Games for Empowerment and Inclusion (IDGEI 2015), At Atlanta, GA*, 2015.

Please contact me if you would like any of the source code or compiled programs used in this thesis.